Foundations of Blockchains
Notes

Tim Roughgarden

Contents

Chapter 1

Introduction and Overview

At first, let's review the goals of this course:
The main focus of this course is the science and technology of blockchain protocols and the applications built on top of them. Therefore we will mainly discuss the fundamental principles of blockchain design and analysis.
It's worth recognizing that we're currently in a particular moment in time, witnessing a new area of computer science blossom before our eyes in real time. It draws on well-established parts of computer science (e.g., cryptography and distributed systems) and other fields (e.g., game theory and finance), but is developing into a fundamental and interdisciplinary area of science and engineering in its own right.

1.1 Overview

Here we start with the phrase "blockchain stack" in which stack refers to a bunch of layers each building on top of the previous one. To explain the organization of these lectures, it's useful to keep in mind a cartoon version of a "blockchain stack" that comprises a number of layers (Figure 1.1). Starting from the bottom and moving on up:

- **Layer 0:** For us, layer 0 will basically be the Internet. That is, it provides at least a semi-reliable. Therefore we are not assuming that the internet is perfect and we will be taking into account all the delays and possible attacks. method for point-to-point communication between untrusted parties.
- **Layer 1:** Layer 1 is the consensus layer, and its job is to keep a bunch of computers (potentially scattered all over the globe) in sync, despite possible network failures and attacks. For example, Bitcoin and Ethereum are both layer-1 protocols. For smart contract platforms like Ethereum, it can also be useful to separate out the compute layer, with the consensus layer merely deciding which instructions (smart contract function calls, etc.) should be executed and in what order, and the compute layer responsible for actually carrying out those instructions and updating the global state. (E.g., full nodes running the Ethereum protocol participate simultaneously in both consensus and compute. It means that because we're participating in a consensus protocol so you will be kind of working hard and talking to other full nodes so that you all stay in sync but what it is you're staying in sync on in a platform like that is a sequence of instructions. so the consensus part is you come to agreement on the sequence of instructions that need to be carried out but then there's still the ta*sk* of actually carrying out that computation and again if you're running a full node in ethereum you're also responsible for the computation that you've come to consensus on so that's a sort of concept two conceptually distinct responsibilities and so sometimes it's worth uh sort of formally differentiating between them)
- **Layer 2:** For us, layer 2 will be the scaling layer. The goal here is basically to implement the same functionality exported by a layer-1 protocol, but a lot more of it. So, layer two is really necessary when you're not happy with the sort of amount of processing power you have at your layer one. For example, Bitcoin and Ethereum can only process so many transactions (roughly 5 per second and 15-20 per second, respectively), and the point of a layer-2 protocol is to scale this capacity up by at least a couple of orders of magnitude.
- **Layer 3:** Finally, on top there is an application layer (as there is in the Internet stack), which refers to the applications built on the functionality provided by the previous layers. (Decentralized exchanges

like Uniswap and NFT marketplaces like OpenSea are examples you might be familiar with.) Here again we're grouping together two logically distinct things, the actual smart contracts that live in the blockchain (sometimes called the "protocol layer") and the user-facing icing on top (e.g., a Web interface). For example, Uniswap is really two things, its smart contracts and its Web interface to interact with those contracts. These are different things: For example, you can interact with the Uniswap contracts directly (as one would via a function call from a different contract, for example) rather than going through the standard Web interface.

Here is a cartoon version of the "blockchain stack" to explain the organization of this lecture series:

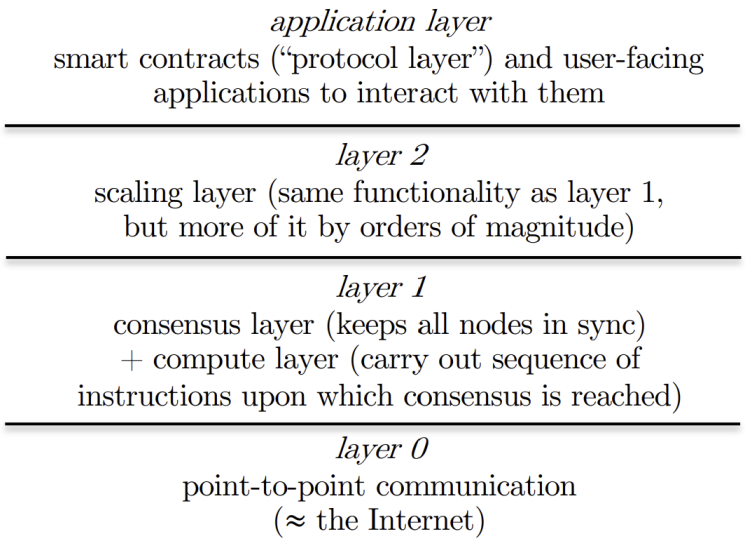


Figure 1.1: cartoon version of the "blockchain stack"

Again, don't take this taxonomy too seriously—it's in flux and not at all standardized. For example, "layer 2 solution" often means something more specific than an arbitrary scaling solution (as we'll discuss in a future lecture). Also, the clean picture of a stack is misleading, as the boundaries between layers are porous. Ideally the layers would be insulated from each other, with protocols for one layer depending only on the provided functionality of the layer below (and independent of the latter's implementation). Currently this is far from true, unfortunately; for example, if you're implementing a decentralized exchange at the application layer, it's really useful to know exactly how the layer-1 consensus protocol works.

1.2 Outline

We can describe the arc of this lecture series in terms of the layers above. We'll *sk*ip layer 0 and assume that we have the functionality of the Internet (such as it is). There's plenty of interesting work about blockchain-friendly layer-0 protocols (e.g., designing a peer-to-peer gossip protocol to make denial-of-service attacks harder), but it's outside our scope. The majority of the lectures—perhaps 60% of them—will be about layer-1 protocols.we'll discuss classical consensus protocols (and how they inspired Tendermint), Nakamoto/longest-chain consensus (one of Bitcoin's key innovations), proof-of-work and proof-of-stake sybil-resistance mechanisms, difficulty adjustment, and "selfish mining." The last four or so lectures on layer-1 protocols will be a deep dive on the world's biggest two blockchains, Bitcoin and Ethereum. we'll learn quite a bit about how these two protocols work—both because it's useful to know, and because it's a prerequisite for understanding how layer-2 scaling protocols work because the layer two solutions sort of necessarily depend on some of the details of how the layer one consensus protocols operate.

Speaking of which, the next 20% or so of the lectures will be about layer-2 protocols (an extremely active area in 2021–2022). We'll discuss the Lightning network, the principal solution thus far for scaling up Bitcoin, and "rollups" (both "optimistic" and "zk/validity"), which appear to be the way forward for scaling up Ethereum.

We'll also cover some newer layer-1 protocols that strive for high throughput out of the box, potentially obviating or at least delaying the need for layer-2 solutions. These are newer generations of layer one consensus protocols that claim to be so much faster than bitcoin and ethereum that they may not need a layer 2 protocol at all.
The final 20% of the lectures will focus on the application layer, and primarily on "DeFi" (stands for decentralized finance), which is where a lot of the action has been over the past couple of years. We'll have a couple lectures on DeFi primitives (stablecoins, price oracles) and a couple on applications built on top of those primitives (borrowing and lending, trading via automated market makers). We'll also talk about "MEV" (for "miner extractable value"), which is a great case study of the current lack of separation between the consensus and application layers.

If we have time, we would be talking about dows so decentralized autonomous organizations and in particular different methods one can take to on-chain governance which is also a very tricky design problem so for example if you want to have a bunch of people vote in a decentralized way on whether or not to take some decision you know what's the best way to implement that kind of on-chain voting super interesting problem which is not very well understood at this point.

1.3 3 comments about this course

A new computing paradigm: For us, blockchains are not about cryptocurrencies or payments per se. They're about a new computing paradigm—a programmable computer that lives in the *sk*y, that is not owned by anyone (or rather, is owned by thousands of people all over the globe, including yourself if you like) and that anyone can use. (There might be a usage fee you have to pay, but there's no access control—you don't need anyone's permission.)

Not about digital money: Unlike most Blockchain courses out there which talk about what money is and teach you the store of value, unit of account, and medium of exchange, in this course, cryptocurrency is the means, not the ends—a tool that helps us implement the functionality that we really want, by charging for blockchain usage and/or rewarding actors that contribute to the protocol and keep it running.

Principles over protocols: We won't start by explaining how one specific blockchain protocol like Bitcoin or Ethereum works. (Though we will learn a lot about those two protocols later.) Rather, we'll start with the fundamental principles of consensus protocol design and analysis (safety, liveness, etc.), and will then understand specific protocols through the lens of these principles.

1.4 Digital Signature Schemes

We're going to begin with the meaning of *consensus*: Consensus roughly means keeping multiple machines in sync (in this context machines are referred to as nodes) despite failures and attacks. Two things are going to make you know consensus quite non-trivial to achieve.
First of all there's a certain unreliability of a communication network like the internet; you're going to have unexpected network delays network outages maybe even sort of malicious attacks like denial of service attacks.
The second thing that makes consensus hard is we don't want to assume that every node is correctly running the intended protocol so maybe some nodes have sort of an out of date or buggy version of the protocol, and maybe some nodes are actually even controlled by a malicious actor who's trying to subvert the protocol.

The permanent assumptions:

1. The Internet exists. (That is, there is a semi-reliable mechanism for point-to-point communication between untrusted parties.)

2. Cryptography exists. For the most part, we won't need anything too exotic, primarily the existence of cryptographic hash functions (discussed in a later lecture) and secure digital signature schemes

1.4.1 Definition of a Digital Signature Scheme

A digital signature scheme is defined by three (computationally efficient) algorithms:

1. **Key generation algorithm:** takes as input a random seed r, and returns a public key-private key pair (pk, sk). As the terminology would suggest, sk should be kept private (it will let you sign digital documents) while pk should be posted in public view (it allows anyone to verify your signature). The two keys are inextricably linked— indeed, typically pk can be derived directly from sk. [In a typical implementation, the algorithm might well take no input and generate its own random seed. E.g., type in ssh-keygen (with no arguments) at a Unix command line.]

2. **Signing algorithm:** takes as input a message m and a private key sk, and returns a signed version of the message (m, sig). (Here sig denotes bits that are appended to the end of the message. The signature length is independent of the message length, and in a blockchain context is typically 520 bits or thereabouts.) Note that the signature depends on the message (and on the identity of the signer, i.e., the provided private key). This is totally different from IRL signatures—your pen-and-paper signature is the same, no matter what the document contents. This property is obviously necessary for digital signatures—a document-independent signature could be easily copied and pasted to forge other signed documents.

3. **Verification algorithm:** is responsible for checking whether or not an alleged signature actually is valid. It takes as input a message m, someone's public key pk, and an alleged signature sig of m by the person who knows the corresponding private key sk. The algorithm answers "yes" or "no," according to whether the signature is valid (i.e., whether or not running the signing algorithm with message m and the private key sk corresponding to pk really would generate the signature sig).

1.4.2 Definition of "Security"

In all of these lectures we'll make a sort of extreme assumption which is still a very close approximation to reality that signatures are ideal. Now what does that mean?

Assumption ("ideal" signatures): it is impossible to forge a signature without knowing the private key, even if you've seen a huge collection of examples of messages that have been signed with that key (with the example messages potentially chosen by the would-be attacker). That is, given such examples and a new (previously unseen) message m, without knowledge of the private key sk, it is impossible to generate a signature sig such that the verification algorithm would respond "yes" to the input (m, sig, pk) (where pk is the public key corresponding to sk).

This assumption is a good approximation of reality (assuming you use a well-implemented digital signature scheme with an appropriate key length). But as a theorist, it's my duty to point out that, strictly speaking, the assumption as stated is false.

Brute Force Attack:
For example, it's possible in principle to break a signature scheme by brute-forcing the private key—given a message m and signature sig generated by the private key sk, an adversary could enumerate all possibilities for sk and try each one, waiting until it manages to regenerate the signature sig (at which point the adversary knows that its current guess actually is sk), and then using the reverse-engineered private key to sign any other messages that it wants. Why doesn't this caveat bother us? Because, assuming the key length l is at least several hundred bits, this brute-force attack would need to enumerate over 2^l possibilities and would be completely unimplementable—the search literally wouldn't finish before the collapse of our sun. (For reference, the estimated number of atoms in the known universe is something like 2^{265}.) Turning this into a theorem thus requires a (modest) assumption, that everybody (including our adversaries) are computationally bounded. For example, we could assume that there exists some polynomial function p (say, $p(x) = x^{20}$ or $p(x) = x^{100}$) such that no adversary can perform more than $p(l)$ computer operations, where ' denotes the key length. (Every polynomial function is asymptotically smaller than every exponential function, so this assumption precludes brute-force search for sufficiently long keys.)

Shortcut Algorithms:
We're still not done, as who said that an adversary won't do anything more clever than brute-force search? When you study algorithms, you learn tons of problems for which bruteforce search would take an exponentially long

time and yet clever algorithms can cut through the clutter and identify a solution in polynomial (often, near-linear) time. (The single-source shortest-path problem and the minimum spanning tree problem are two classic examples.) Who's to say a clever adversary can't find a clever short cut and reverse engineer a private key sk from a collection of signed messages much faster than brute-force search? In response, we need to make more assumptions—called (computational) complexity assumptions or hardness assumptions—that certain problems cannot be solved efficiently (meaning in time polynomial in the input size).5 For the digital signature schemes most commonly used in blockchain protocols, security is based on the assumption that there is no polynomial-time algorithm for the discrete logarithm problem in a suitably chosen group (i.e., the problem of reverse engineering the exponent x from the terms g and g^x , where g is a group generator and g^x denotes g multiplied by itself x times). Finally, we need to deal with the fact that an attacker may use a randomized algorithm, and so in principle (with nonzero but extremely low probability) might get lucky and randomly guess our key. For this reason, any formal statement must tolerate a nonzero but negligible failure probability. Thus, the formal statement of the security of a digital signal scheme would look something like this: assuming a polynomial-bounded (randomized) attacker and suitable complexity assumptions (like the hardness of discrete log), an attacker that knows a collection of messages signed with the private key sk has only a negligible probability of generating a valid signature (the one that would be generated by signing with sk) on a new message m. (The example messages can be chosen by the attacker, and can depend on m.) Security statements of this form have been proved (under the stated assumptions) for the digital signature schemes used in practice.

1.4.3 Digital Signatures in Consensus Protocols

You can probably see why digital signatures are important for various blockchain use cases (e.g., signing off on a transfer of your funds), but they can also be tremendously useful in the design of the underlying consensus protocol. For example, they prevent a node A from credibly claiming to a node B that a third node C sent a message m to A some time in the past—as long as all the nodes are signing their messages, B will only believe A if A can exhibit a version of m that has been signed by C. With digital signatures, they can't make up fake messages from other nodes; all nodes can do is repeat without modification what they've heard. Unless otherwise noted, when we discuss consensus protocols, we will assume that every message sent by one node to another is signed by the sender.

1.5 The State Machine Replication (SMR) Problem

There are several notions of "consensus"; we'll see at least three. We'll start with the version most immediately relevant to blockchain protocols, called the state machine replication (SMR) problem. What's a "state machine"? If you've ever studied automata (e.g., deterministic finite automata (DFAs)) you have a good sense of what it means (states and a state transition function). If not, some examples might help. One of the old-school applications of the SMR problem is managing a replicated database. Think of a big company like IBM, with some database of valuable information. Suppose they want to charge customers for access to it (e,g., via queries and updates), but also want to promise 99.999% uptime. You won't get that level of uptime if the database is stored on a single computer (hardware and software failures being too common). An obvious idea for boosting uptime is to have multiple copies of the database, with each copy stored on a different machine and in a different location (so that machine failures are somewhat independent). But as soon as you have two or more copies of the data, you've got a new problem—keeping them in sync with each other. (E.g., if a customer writes an update to one copy, the update must also be reflected in the other copies, so that a corresponding read returns the same answer no matter which copy you ask.) Here, "state" means the current contents of the database, and each write to the database would effect a "state transition," moving from one state to a new one (which reflects the new write operation). In a blockchain context, "state" will refer to the current status of the blockchain and its users (e.g., the current balance of each account, the local state managed by smart contracts, etc.). Executing a transaction (e.g., a payment from one account to another) effects a state transition (e.g., with the new state reflecting the post-transfer account balances). Unlike in the database example, where the only reason for replication is to increase uptime, in a blockchain context, the primary motivation for replication is "decentralization," meaning to ensure that responsibility for the protocol is distributed over many machines, with no one actor having significant control over its state and execution.

1.5.1 Problem Definition

In both the database and blockchain examples, the goal is to keep a bunch of nodes in sync, meaning all of them make the same sequence of state transitions (database operations/transaction executions) and hence agree on the current state of the state machine (database contents/blockchain state). This is the SMR problem. Summarizing:

1. There is a set of *nodes* responsible for running a consensus protocol, and a set of clients who may submit "transactions" to one or more of the nodes.

2. Each node maintains a local append-only data structure—an ordered list of transactions that only grows over time—which we'll call its history. What we like to see happen is for all of these nodes to have identical local histories because we wish to keep them in sync despite potentially failures delays and attacks.

Note that order is quite important. If two writes to a database conflict, it matters which one is carried out first. In a blockchain context, if two submitted transactions spend the same coins but with two different recipients (an attempted "double-spend"), it matters which transaction is executed first (as the second one will fail on the grounds of insufficient funds). Informally, the goal in the SMR problem is to deploy code that keeps all the nodes in sync, with the same local histories (same ordered sequences of transactions). But what does this actually mean?
First, what form would a "solution" to the SMR problem take? Answer: a protocol.
We won't bother with an overly formal definition, but think of a protocol as a piece of code that is to be run by each of the nodes. It's going to be basically a bunch of functions and when one event happens that will trigger one of those functions, which will then do some stuff. This code manages both the computations and the communications performed by the node as the protocol runs. Specifically, each node can:

- maintain local state, and perform local computations that depend on or affect that state
- receive messages from other nodes and from clients
- send messages to other nodes

The code is event-driven, meaning that when some event occurs (e.g., receiving a new message from a client or another node), it can trigger a response from the node (e.g., some local computations followed be sending out new messages to one or more other nodes).

Given such a protocol when should we say that it "solves" the SMR problem? It's a lot harder to define a correct answer for these algorithms because these protocols are sort of running forever and it's not like there's some single output at the end so we need to think a little harder about what it means to correctly solve a consensus problem like SMR. What does it mean for a protocol to be a "correct" solution to the SMR problem? That is, what guarantees do we want from a protocol? We can distinguish between safety guarantees, which promise that a certain bad event never happens, and liveness guarantees, which promise that a certain good event eventually happens. We'll focus on one of each.
Goals:

1- Consistency (safety property): We say that a protocol satisfies consistency if all the nodes running it always agree on the history (i.e., the same ordered transaction sequence). Actually we'll be a little more flexible—if there's a node in Siberia that always finds out about the latest transactions later than everyone else, it's OK if its local history lags, as long as it's always a prefix of other nodes' histories (i.e., just needs to catch up). What absolutely cannot happen is for two nodes to disagree on the relative order of two different transactions. Consistency is the safety property promising that this bad event never occurs. If we only cared about consistency, our lives would be easy. After all, the empty protocol (with all nodes maintaining an empty history forevermore) satisfies consistency! So we also need a guarantee that work eventually gets done.

If the only thing we cared about was consistency in this sense consensus or SMR would not be a hard problem because for example a really easy way to make sure that all the nodes always have the same history is to never add anything to anybody's history and literally do nothing. In this case, all the nodes will always have the empty sequence and we'll always be in sync which obviously is not what we have in mind so we need to impose another constraint on the protocol to qualify as a solution and this is going to be the liveness property.

2- Liveness: Every valid transaction submitted to at least one node is eventually added to every node's local history. The meaning of valid here varies depends on the details of the blockchain we're dealing with for instance if it's a currency, it should be signed by the sender and there should be sufficient currency in that person's account. Are there protocols that solve the SMR problem, in the sense of satisfying both consistency and liveness? As we'll see, the answer depends on a number of factors, including the reliability of the underlying communication network and the number of compromised nodes. In the following lectures you'll learn the key possibility and impossibility results for SMR consensus. We'll eventually see how these theoretical results give us a lens through which to compare different layer-1 protocols (e.g., some of which favor liveness, others of which favor consistency). Next lecture, we'll assume a super-reliable communication network and give a protocol that solves the SMR problem even in the face of an overwhelming number of compromised nodes. Later lectures will discuss protocols that solve the SMR problem under weaker assumptions about the communication network (but stronger assumptions about the number of compromised nodes).

Overview of Lectures 2-7

Let's see what we have in the upcoming lectures:

- **Lecture 2:** Synchronous model. In these models we assume that messages are delivered in a reasonable amount of time. There will be a bound Δ on the number of time steps it can elapse before that message is guaranteed to have been received. This is a semi-reasonable model of a communication network in times of normal operation. For example for the best case operation of the internet, the Δ is assumed some where near a second.
  The good news about the synchronous model is that it enables some pretty impressive SMR protocols. The main goal of Lecture 2 is to explain the design and analysis of such a protocol, known as the Dolev-Strong protocol (from the early 1980s). This protocol is for a single-shot consensus problem known as "Byzantine broadcast," but we'll also see in Lecture 2 how to reduce the SMR problem (which is multi-shot consensus) to the Byzantine broadcast problem. This reduction extends the Dolev-Strong protocol to an SMR protocol with all the properties that we might want. What's really extraordinary about the DolevStrong protocol is that it tolerates an arbitrarily large number of dishonest nodes. Even if 98 out of 100 nodes are malicious and acting in cahoots, there's nothing they can do to trick the remaining 2 honest nodes into getting out of sync. The disadvantage of the synchronous model is that its assumptions are simply too strong to provide accurate guidance for the design of Internet-scale protocols. You won't see the Dolev-Strong protocol referenced very often in blockchain whitepapers and discussions, but it's worth studying for several reasons. One, if you ever do find yourself in a setting where the synchronous model is appropriate (perhaps in a private network), you'll know how to achieve a pretty amazing set of guarantees. Second, studying this (relatively simple) protocol will act as a good warm-up to more complicated protocols that we'll see later (like Tendermint)—we'll get a sense for what typical consensus protocols look like and how they can make use of tools such as digital signatures. Finally, it's a famous result— definitely part of the distributed computing greatest hits reel—and readers who identify as card-carrying computer scientists should enjoy learning how and why it works.

- **Lecture 3:** Synchronous model, Lecture 3 is where we see our first impossibility result which, like the one in Lecture 6, rules out good SMR protocols when at least 33% of the nodes can deviate from the protocol (or in other words 67% honest nodes are needed). Unlike the Lecture 6 impossibility result (for the partially synchronous model), the one here applies even in the (easier) synchronous model. But wait, why doesn't this contradict the guarantees promised by the Dolev-Strong protocol, which hold even when almost all the nodes are malicious? This lecture drills down on a key assumption made by the Dolev-Strong protocol, typically called the PKI (for "public key infrastructure") assumption. Here we're not just assuming that cryptography exists and that anybody has access to a secure digital signature scheme but we're also assuming that prior to the commencement of the protocol, all the nodes' public keys are common knowledge. The assumption is that all the nodes running the protocol have their own public key-private key pairs and that all nodes' public keys were somehow exchanged before the commencement of the protocol. In other words, all nodes begin the protocol with the ability to verify a signature by any other node. While the PKI assumption is fairly palatable in a blockchain context, it winds up being quite interesting to study what's possible without it and other "trusted setup" assumptions. And the 33% impossibility result of this lecture pops up already in the synchronous setting when there are no trusted setup assumptions. The proof of this impossibility result—due to Fischer-Lynch-Merritt and called the "hexagon argument," for reasons that will become clear—is impressively slick. For those of you who like cool proofs, Lecture 3 is likely to be your favorite one of the bootcamp. Conversely, if you're not into proofs and wanted to skip one of these six lectures, Lecture 3 should probably be the one. It's a famous and enlightening result that belongs to

the distributed computing canon (like the other impossibility results in the bootcamp), but it is not 100% essential to the bootcamp's overarching narrative.

- **Lecture 4:** Asynchronous model, The synchronous model allows consensus protocols with amazing guarantees (at least under the PKI assumption) but makes assumptions that are simply too strong for protocols that operate over the Internet. We therefore have no choice but to weaken our assumptions about the underlying communication network. One natural idea is to assume nothing about the communication network, other than the minimal property that every message is delivered eventually. This is the idea behind the "asynchronous model." The good news about the asynchronous model is that, with essentially no assumptions about the communication network, any consensus protocol with non-trivial provable guarantees in the model would automatically be impressively robust. The bad news is that this model throws out the baby with the bathwater, in the sense that consensus is provably impossible (at least for deterministic protocols). This result, due to Fischer-Lynch-Paterson, is perhaps the most famous impossibility result in all of distributed computing. Its proof will be the hardest one that we'll see in this bootcamp, and probably the hardest one in the whole lecture series. We'll get started on the proof in Lecture 4 (after formally defining the asynchronous model) and finish it off in Lecture 5. The key takeaway from the FLP impossibility result is that the asynchronous model simply doesn't make enough assumptions—for us, the point of a model is to help us brainstorm about the space of protocols and hopefully identify some that work well in both theory and practice. With no good (deterministic) protocols in this model, there's not much brainstorming to be done! The FLP impossibility result thus completes the justification for introducing a third model (the partially synchronous model) in Lecture 6; while slightly less natural than the synchronous or asynchronous model, it acts as a sweet spot between the two, with assumptions that are weak enough to be relevant to Internet-scale protocols and strong enough so that practical consensus protocols with provable guarantees actually exist.

- **Lecture 5:** Asynchronous model,

- **Lecture 6:** This lecture will be about partially synchronous model. The main idea is to look at the operation of a protocol in 2 different phases: 1- Normal operations where messages get delivered in a reasonable amount of time and 2- Attack mode: situation where there's big network outages and attacks. Synchronous model says that you don't want to break too badly under attack and that you recover quickly when the attack ends. The partially synchronous model is quite important in analysis of blockchain protocols. This lecture introduces the partially synchronous model as a sweet spot between the synchronous and asynchronous models, proves that consensus is impossible in this model when more than 33% of the nodes can deviate from the protocol, and compares these results to a famous principle from distributed systems, the CAP theorem.

- **Lecture 7:** We're going to talk about the Tendermint protocol. In this lesson we're mainly focusing on principles rather than protocols but lecture 7 will be about a specific blockchain protocol that's used in the real world and we will discuss its provable guarantees which include consistency and "eventual" liveness. Through these proofs not only we understand how Tendermint works but also why it works the way it does

If we look at the main layer one of the protocols we can see that most of them fall into two camps:

1. **BFT-type protocols** We will refer to blockchain protocols that resemble the consensus protocols from the 1980s and 1990s (like Tendermint) as BFT-type protocols. (Here BFT stands for "Byzantine fault tolerant.") Several of the major "layer-1" networks use BFTtype blockchains, including Solana, Terra, Cosmos, and Algorand. When Facebook/Meta was considering rolling out its own blockchain (for the Diem cryptocurrency), they were planning to base it on a BFT-type protocol called HotStuff. This bootcamp thus prepares you well to understand how these BFT-type blockchain protocols work. In this type, failure is mostly caused by progress stalls (no new blocks are created) which happens because this type of protocols favor safety to liveness.

2. **Longest-chain protocols:** We haven't mentioned the two most famous blockchain protocols of them all, Bitcoin and Ethereum. These two protocols achieve consensus in a radically different way and are examples of longest-chain protocols. The idea of longest-chain consensus does not come from the classical consensus literature and was invented specifically for the Bitcoin protocol. (Several blockchain protocols that came later, including Ethereum, also adopted longest-chain consensus.) Our first order of business after our

classical consensus bootcamp is to develop a deep understanding of longest-chain consensus and its pros and cons relative to BFT-type consensus. We'll get started on this topic in Lecture 8. In this type, for example for bitcoin or Ethereum, we don't really have stopping or stalling. So the problem isn't caused by progress staalls but there's a massive rollback of a bunch of transactions in favor of an entirely different batch of transactions. This is because this type of protocols favor liveness to safety.

At the end of these lectures, you will have accomplished two really important things. First, you'll have mastered the basic concepts behind one of the two major design paradigms for blockchain protocols (BFT-type protocols). Second, and possibly even more important, you'll be equipped with the mental model, the formal definitions, and the language to discuss, assess, and rigorously compare different blockchain consensus protocols (including longestchain protocols and anything else that protocol designers manage to come with).

What's even more important to remember is the lens through which we look at these different parts of the design space and assess the different trade-offs that they're making. For example some type of blockchain favors saftey like BFT-type protocols and some favor liveness like Longest-chain protocols. So, by mastering the partially synchronous model and the impossibility results therein, we can actually know what people mean when they say one blockchain favors safety and the other favors liveness. Also, we can understand why different blockchain protocols fail in different ways

Chapter 2

Byzantine Broadcast in the Synchronous Model via the Dolev-Strong Protocol

2.1 Recap of SMR Problem

The goal of this lecture is to design and analysis a consensus protocol that solves the state machine replication (SMR) problem under a strong assumption about the underlying communication network (known as the "synchronous model" assumption), meaning that the protocol is guaranteed to satisfy both consistency and liveness. Recall the definition of the SMR problem from Lecture 1:

1. There is a set of nodes responsible for running a consensus protocol, and a set of clients who may submit "transactions" to one or more of the nodes.
2. So what are nodes doing besides just receiving and transactions from clients? Each node is responsible for maintaining a locally stored history. Each node maintains a local append-only data structure—an ordered list of transactions that only grows over time—which we'll call its history.

For us, "clients" represent users of a blockchain protocol, "nodes" refer to the machines actually running the protocol, and a "transaction" would be something like a cryptocurrency transfer or a smart contract function call.

As we said in the previous lecture, order is very important to us. For example if we think about two transactions that are currency payments both trying to spend the same coins sort of an intended double spend attack perhaps it really matters which of those two transactions comes first in the history the first transaction will succeed and the coins will be spent when the second transaction comes along the coins will already be spent and that transaction will fail so they maintain the transactions that have been executed along with the order in which they have been executed.

Recall also that a protocol is, informally, a piece of code that is to be run by each of the nodes to manage the local computation at and communication by the node. For the SMR problem, we're looking for a protocol that satisfies two properties, one a safety property (bad things never happen) and one a liveness property (good things eventually happen)

Goal 1: Consistency: We say that a protocol satisfies consistency if no two nodes ever disagree on the relative order of two different transactions. (Ideally they would stay perfectly in sync, but we want to allow some nodes to fall behind as long as they eventually catch up with the others.)

Goal 2: Liveness: Every transaction submitted to at least one node is eventually added to every node's local history. (For now, think of all transactions as always being valid and eligible for inclusion.)

Satisfying either of these two properties along is trivial (why?); what's hard is getting both at the same time. So do there exist SMR protocols that satisfy both consistency and liveness? Over the next few lectures, we'll learn that the answer to this question depends in interesting ways on what assumptions you make, for example on the reliability of the communication network, the fraction of malicious or otherwise corrupted participants, and whether or not any "setup" is allowed in advance of the protocol's commencement. In this and Lecture 7, we'll see possibility results, which identify assumptions under which the answer is yes (and provide a concrete SMR protocol that proves it). These two lectures sandwich a number of impossibility results, which identify assumptions under which the answer is "no" (and provide mathematical proofs of that fact).

2.2 Initial Assumptions

In the next several lectures, we are going to make a bunch of assumptions, really more assumptions than we're comfortable with. It will then be pretty easy to see that there are protocols satisfying liveness and consistency under these assumptions. Then we'll work hard to relax those assumptions one by one, leading to more sophisticated protocols that are more robust solutions to the SMR problem. Here are four assumptions that will make the SMR problem quite easy to solve.

2.2.1 Assumption 1: Permissioned Setting

The first assumption is that the set of nodes responsible for running the protocol is fixed and known upfront. That is, the protocol description itself can reference the specific nodes that are going to be running it. (And because the protocol is deployed at every node, every node then knows about every other node.) We'll use n to denote the number of nodes. The nodes have distinct (and a priori known) names; without loss of generality, those names are $\{1, 2, 3, ..., n\}$. Similarly, they have known and distinct IP addresses (and hence can communicate with each other). So when a node sort of spins up and fires up this protocol it's actually given a list of n minus one IP addresses and basically the pro basically being told these are the $n-1$ other IP addresses you should be expecting to hear from to receive messages from uh and you should feel free to send those other $n-1$ nodes your own messages as well.

Nearly the entire 20th-century literature about consensus protocols works in the permissioned setting, because that was a perfectly reasonable assumption for the motivating applications at that time. For example, if IBM wanted to replicate a database seven times in order to achieve very high uptime, they would simply buy seven machines and then run a consensus protocol on this (a priori known) set of machines.

To discuss blockchains (and Bitcoin and Ethereum in particular), we'll eventually want to graduate from the permissioned to the permissionless setting. But there are a number of reasons to start with an in-depth study of the permissioned setting. For example, all the impossibility results that we'll be proving for the (easier) permissioned setting will apply automatically to the (harder) permissionless case. Similarly, when brainstorming about a permissionless protocol, it can be useful to first tackle the permissioned setting as a special case (what would you do if you knew all the nodes up front?), and then bootstrap it to the general case. Indeed, several high-profile blockchain protocols follow this approach, effectively implementing a reduction from permissionless consensus to the permissioned consensus.

2.2.2 Assumption 2: Public Key Infrastructure (PKI)

You can think of the PKI assumption as an extension of the permissioned assumption—not only do all the nodes know about all the other nodes (via their named and IP addresses), but all the nodes also have distinct public-private key pairs, with all the public keys common knowledge at the start of the protocol. (The private key of a node is known only to that node itself.) Thus, every node begins the protocol in a position to verify signatures by all the other nodes (by running the verification algorithm specified by the digital signature scheme). The PKI assumption is stronger than assuming merely that cryptography exists—it also requires that all the nodes somehow shared their public keys with each other. This is an example of a trusted setup assumption, in that it asserts that a certain computation (in this case, public key distribution) is done correctly in advance of the protocol's commencement, remaining silent on how this computation might actually happen.

You can probably imagine various ways of approaching the problem of public key distribution, but here we're just going to take it on faith that it happened. Of the four assumptions in this section, the PKI assumption might bother us the least, and we won't really focus on relaxing it (unlike the other assumptions)—if the biggest flaw with your protocol is that it requires public key infrastructure, it's probably a pretty good protocol. That

said, some blockchain protocols (including Bitcoin and the initial version of Ethereum) do not require the PKI assumption.

2.2.3 Assumption 3: Synchronous Setting

A crucial assumption in this and the next lecture is that we're going to make a very optimistic assumption about the behavior of the underlying communication network—formally, that protocols operate in what's known as the synchronous model. You can think of this model as making two sub-assumptions.

Shared global clock: The first, which maybe we could live with, is that all of the nodes share the same global clock. That is, even without any communication, all the nodes always agree on exactly what time it is. If we break time into time steps, such as intervals of ten seconds, all nodes automatically agree on what time step they're currently in. This sub-assumption is not literally true in the real world (for example, due to clocks drifting at different rates), but you can start imagining ways that you might approximate it in practice.

Bounded message delays: The second sub-assumption is the one that should bother us a lot: totally reliable delivery of information across the communication network. Specifically, we'll assume that every message sent by one node to another at the beginning of some time step t arrives at the intended recipient prior to the beginning of time step $t+1$. For example, if time steps correspond to 10-second time step, messages sent at the 80-second mark are all guaranteed to arrive before the 90-second mark. The model makes no other assumptions about the order in which a node receives messages (e.g., messages that arrive in the same time step might arrive in any order). If your communication network is the Internet, this second sub-assumption might hold in the best-case scenario of "normal operating conditions" (assuming a generous time length, like 10 seconds), but it's completely unreasonable if you're worried about network outages (which of course happen all the time in the Internet) and denial-of-service attacks (which should certainly be expected if your protocol secures billions of dollars of value).

The synchronous baseline: When probing the guarantees that a blockchain protocol offers, the synchronous model serves as a useful sanity check. A necessary condition for a good blockchain protocol is that it works really well in the synchronous model—with minimal other assumptions (e.g., on the fraction of corrupted nodes), it should guarantee consistency, liveness, good efficiency, etc. (And this will be the case for the Dolev-Strong protocol described in this lecture.) But you can't stop there, as real-world blockchain protocols really should be robust when the communication network in unreliable (e.g., due to a denial-ofservice attack). We'll see in Lectures 4–5 that in this case, you can't have it all—when under such an attack, every blockchain protocol must give up consistency or liveness. So, when you're assessing a blockchain protocol, it's your duty to ask how it handles the stress test of a prolonged network outage or denial-of-service attack. Does it give up liveness? Does it give up consistency? God forbid, is it a badly designed protocol that gives up both? These are the questions you need to ask when you're comparing different consensus protocols

2.2.4 Assumption 4: All Honest Nodes

The final assumption is a ridiculous one, and we'll start relaxing it in later in this lecture. But just for the next section, let's assume that all of the nodes running the protocol are honest. Here "honest" is actually a description of nodes' behavior, not of their (owners') intent, and means that the all the nodes run the intended protocol, correctly and without deviations or bugs. This assumption is way too strong even for those old-school applications from the 1980s. For example, suppose IBM is running seven servers, each with a copy of a database. Once in a while, one of those servers is going to go down, unable to continue following the protocol (thus qualifying as "dishonest").

In the next section we'll get our feet wet by designing a consistent and live SMR protocol under the all-honest assumption. The rest of this lecture develops a more complicated SMR protocol that, under the first three assumptions above, satisfies consistency and liveness no matter how many nodes stray from the protocol.

2.3 Solving SMR via Round-Robin Leaders

A lazy SMR protocol. Perhaps the laziest SMR protocol is the one in which nodes never bother to communicate, which each node independently adding transactions to its local history as it hears about them. This trivial protocol fails to solve the SMR problem even under all four of the assumptions in Section 2.2. If every transaction submitted by a client always arrives at exactly the same time at every node, then we'd be OK. But if a client only submits a transaction to a subset of the nodes, or if network delays cause transactions to arrive at different nodes in different orders (which is possible even in the synchronous model), then consistency will generally be violated.

Coordination via rotating leaders. The lazy protocol above highlights the need to coordinate the nodes, so that they're all aware of the same set of transactions (in some canonical order). We'll achieve that coordination through rotating leaders, repeatedly iterating through the nodes in round-robin order. E.g., with 100 nodes with names $\{1, 2, ..., 100\}$, node 7 will be the leader in time step 7, time step 107, time step 207, and so on. It is easy to implement the rotating leaders idea under the assumptions in Section 2.2. Because we're in the synchronous setting, there's a shared global clock and all nodes always agree (without any communication) on what the current time step is. Because we're in the permissioned setting, the set of nodes (and their names) is known in advance and thus every node knows the round-robin order (and particular the time steps in which it is the leader). The leader's responsibility is to coordinate the nodes during that time step:

Note:-

Prescribed Actions of a Leader Node:

1. Collect together all the not-yet-included transactions that it has heard about and orders them arbitrarily (e.g., in the order in which it heard about them).
2. Send the ordered list of transactions to every other node.

By the beginning of a time step t, because we're working in the synchronous setting, every node has received the list of transactions sent to it by the leader of time step $t-1$. At this moment in time, each node (including the leader of time step $t-1$) is instructed by the protocol to append this list to its local history. So that's the protocol. Nodes keep track of when they are the leader and broadcast ordered lists of transactions during those time steps, and also continuously append such lists to their local histories as they hear about them.

Formal proofs of consistency and liveness. Under the four assumptions in Section 2.2, this simple SMR protocol gives us what we want.

Proposition 2.3.1

Under the assumptions in Section 2.2, the protocol above satisfies consistency and liveness.

Proof: Let's start with consistency, the safety property asserting that no two nodes ever disagree on the relative order of a pair of transactions. This protocol satisfies this property because all the nodes operate completely in lock step. At each time step t, the (honest) leader of that time step sends exactly the same list of transactions to every node. By the assumptions of the synchronous model, all these messages arrive prior to the start of time $t+1$. At the beginning of time step $t+1$, all the nodes add these (identical) lists to their local histories. Because nodes start with identical local histories (the empty list), by induction, they remain in sync (have identical local histories) forevermore. What about liveness? Suppose a client submits a transaction to at least one (and possibly only one) node. Because every node is periodically a leader (once every n time steps, where n is the number of nodes), eventually, a node aware of this transaction becomes the leader of a time step. At that point, the (honest) leader will include the transaction in the transaction list that it broadcasts to all the other nodes. The argument for liveness, that's the point where we really needed there to be a rotating leader okay because maybe not all transactions not all nodes have heard about a transaction so we need to sort of take turns so that the node that knows about a transaction has the opportunity to add it to everybody's local histories 😄

Now that we've gotten our feet wet and have some initial experience with the design and analysis of consensus protocols, let's get serious and tackle the SMR problem without the ridiculous all-honest assumption.

2.4 Faulty/Byzantine Nodes

An honest node is one that never deviates (intentionally or unintentionally) from the prescribed protocol. Nodes that deviate from the protocol (whether by intent or by accident) are called faulty. What's the appropriate model of a faulty node? That is, what types of deviations from the protocol should we consider? This question has been studied extensively in the distributed computing literature. To give some context, this section describes three different models of faulty nodes, in order from most to least benign. The third model (of "Byzantine" nodes), is by far the most relevant one for permissionless blockchain protocols (which, ultimately, will be our focus in this lecture series).

Types of faulty nodes:

- **Crash faults.** A *crash fault* occurs when a node simply stops working at some point in the protocol, as if someone pulled out the plug. In other words, the only deviation from the protocol that we consider is, after some (crash) time t, the node no longer sends or receives any messages (and up to time t it correctly follows the protocol). You can imagine why researchers in the 1980s might have been fixated on crash faults, for instance in our running database replication example (with IBM running seven machines of its own, each with a copy of the database). When hardware failures are the main worry (as opposed to software bugs, a faulty network, or malicious attack), crash faults are the sensible ones to focus on. So, with crash failures you might expect to not make as much progress uh progress as quickly you might expect to lose transactions that were sent only to the crash nodes but otherwise everything's fine in particular you never have violations of consistency uh in the presence of crash faults in that simple rotating leader protocol

- **Omission faults.** A more general type of a fault is an *omission fault*. Here, a faulty node can deviate from the protocol by withholding any subset of the messages that it's supposed to send (but it never makes up phony messages that it's not supposed to send). Omission faults can be the result of bad actors, but they also arise more innocently from network delays. For example, consider a protocol that is designed for the synchronous setting and instructs nodes to ignore any messages that don't arrive on time. Whenever a message is delayed more than one time step by the communication network, that message is effectively omitted (because it is ignored by its recipient). A crash fault is the special case of an omission fault in which, after some moment in time, all future messages are omitted (whereas with an omission fault some but not all may be omitted). Omission faults can send some of the messages but then also not send all of them and actually are slightly more powerful than crash faults because model of a faulty node actually messes up the rotating leader protocol that we had previously. There's no issue with liveness; good things still happen when an honest node winds up being the leader but you can violate consistency if you have a faulty node as the current leader right because with an omission fault the leader may send a non-empty list of transactions to some of the other nodes but then it would withhold those messages from the other part of the nodes so half the nodes would be getting a non-empty set of transactions half the nodes would be getting nothing and they would then commit different things to the ends of their local histories half of them would add nothing to their local histories half of them would add a non-empty set of transactions and that's an immediate violation of consistency the histories are not the same at that point.

- **Byzantine faults.** With blockchain protocols that secure billions of dollars of value, you really can't afford to assume away possible deviations that a dishonest node might think of. A *Byzantine* node is one that can deviate from the protocol in arbitrary ways. Distributing computing researchers defined Byzantine faults in the 1980s even though they weren't particularly worried about malicious actors (e.g., think of seven machines all bought and operated by IBM). Why? Because of possible software errors (e.g., in the database implementation). Unlike hardware failures (leading to crashes) and network delays (leading to omissions), it's completely unclear how to model a node that is running a buggy version of the intended software. To avoid controversy and pursue the most general results possible, researchers explored what can and cannot be done in the presence of Byzantine nodes, decades before blockchains were a gleam in anyone's eye. While Byzantine nodes can in principle throw out the protocol and do whatever they want, you might want to think of their canonical strategy as to send contradictory messages to different nodes. For example, in the SMR protocol in Section 2.3, if the leader is Byzantine, it could send different lists of transactions to different nodes. As with omission faults (the special case in which some nodes all receive the same list and the rest receive nothing), that protocol does not satisfy consistency if there is even a single Byzantine node.

Assumption 4 relaxed: bounded number of Byzantine nodes. Byzantine nodes can ignore your protocol and do whatever they want. A good protocol should allow the honest nodes to achieve the desired functionality (e.g., consistent and live state machine replication) despite the best coordinated efforts of the Byzantine nodes. The more of the nodes are Byzantine, the more difficult this goal is to achieve. The sensible relaxation of our previous "all-honest" assumption is that to assume some bound, denoted f, on the maximum number of nodes that might be Byzantine. Equivalently, this relaxed assumption asserts that at least nf of the n nodes correctly follow the protocol. (The all-honest assumption is the special case of $f = 0$, and our simple SMR problem does not satisfy consistency already when $f = 1$.) You might want to think of $\frac{n}{3}$ and $\frac{n}{2}$ as canonical values of f. The parameter f is assumed to be known up front (and hence the description of a protocol may depend on its value). The identities of the (at most f) Byzantine nodes are not known up front. (If they were, the protocol could simply ignore all their messages and effectively operate in the all-honest setting.) So what makes it tricky is like you know there's some byzantine nodes out there and you don't know exactly who they are. Thus a protocol must work simultaneously for every possible coalition of at most f Byzantine nodes.

2.5 The Byzantine Broadcast Problem

Our simple rotating leaders SMR protocol achieves consistency and liveness when $f = 0$ but not when $f = 1$. To achieve fault-tolerance, we need to come up with a more sophisticated protocol. The good news is that we can keep the rotating leaders idea as-is (with nodes taking turns as leaders, for example round-robin). There will be time steps in which the leader is Byzantine(dishonest), however, so honest nodes can not just naively believe whatever the current leader tells them (as they do in our simple protocol)—intuitively, they should also carry out some "cross-checking" to make sure they don't get tricked into inconsistency. We'll abstract out this cross-checking functionality into a single-shot consensus problem that is interesting in its own right, called the *Byzantine broadcast problem*. In the next section, we'll see that fault-tolerant state machine replication reduces to fault-tolerant Byzantine broadcast—any solution the latter (single-shot) consensus problem can be combined with the rotating leaders idea to solve the former (multi-shot) consensus problem. Formally, in the Byzantine broadcast problem, one node is the *sender* (everybody knows who the sender node is) and the other $n1$ nodes are *non-sender*. (For us, the sender will correspond to the leader of the current time step in a rotating leaders-type protocol.) The identity of the sender is known to all of the nodes in advance (as is the case in the rotating leaders application). The sender additionally has a private input v^*, which belongs to some set V (V is all conceivable ordered lists of transactions a sender might be wanting to send out). (For us, v^* will be an ordered list of transactions, and V will be all possible such lists.) By "private," we mean that when the protocol commences, nobody other than the sender knows anything about what v^* is (other than that it is some element of V). What constitutes a "solution" to the Byzantine broadcast problem? Intuitively, we want honest senders to be able to broadcast their private input to all the honest non-senders, while also foiling a Byzantine sender who wants to trick honest nodes into inconsistency. Formally, we'll insist on three guarantees from a protocol:

Note:-

Desired Properties of a Byzantine Broadcast Protocol:

1. Termination. Every honest node i eventually halts with some output $v_i \in V$. (Informally, v_i is node i's best guess as to what the sender's private input v^* is.)
2. Agreement. All honest nodes halt with the same output (whether or not the sender is honest).
3. Validity. If the sender is an honest node, then the common output of the honest nodes is the private input v^* of the sender.

Agreement is a safety property (playing a similar role as consistency), stating that that no two nodes ever disagree on their outputs (even if the sender is Byzantine). Validity (coupled with termination) is effectively a liveness property, stating that a good event (accurate broadcast of the sender's private input) occurs whenever the sender is an honest node. 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 an honest sender broadcasting their private input to all nodes, and honest non-senders outputting whatever they hear from the sender). As with the SMR problem, what's challenging is designing a protocol that satisfies both the safety and

liveness requirements. Because Byzantine nodes can throw away the protocol and/or their private input and do whatever they want, it doesn't make sense to apply any of these requirements to nonhonest nodes (e.g., they can choose to loop forever), nor does it make sense to require anything other than agreement in the case of a Byzantine sender (a Byzantine sender can undetectably pretend that its private input is something other than what it actually is).

2.6 SMR Reduces to Byzantine Broadcast(BB)

We singled out the Byzantine broadcast problem because any solution to it can be used as a "black box" to solve the state machine replication problem (under the same assumptions, e.g., with the same value of f). The idea behind the reduction is simple: use rotating leaders, an in each iteration invoke a Byzantine broadcast subroutine, with the current leader as the sender.

Note:-

A Reduction SMR to Byzantine Broadcast
Assumptions: synchronous (Section 2.2.3) and permissioned (Section 2.2.1) setting with node set $N = \{1, 2, \cdots, n\}$.
Given: a protocol π for the Byzantine broadcast problem that, when at most f of the nodes can be Byzantine, satisfies agreement and validity and always terminates in at most T time steps.
Reduction:

1. At each time step $0, T, 2T, 3T, \cdots$ that is a multiple of T:
   (a) Define the current leader node using a round-robin ordering. (With node 1 the leader at time step 0, node 2 the leader at time step T, and so on.)
   (b) The leader collects together all the not-yet-included transactions that it has heard about and assembles them into an ordered list L^*.
   (c) Invoke the assumed subroutine π for the Byzantine broadcast problem, with the leader node acting as the sender and with the list L^* as its private input.
   (d) When π terminates, every node i appends its output L_i in the Byzantine broadcast problem to its local history.

This reduction is well defined—handed a protocol for Byzantine broadcast on a silver platter, it builds a protocol for state machine replication. Because there is a shared global clock (one of the assumptions in the synchronous model) and an a priori known set of nodes, all nodes automatically know which node is the current leader. Because π terminates within T time steps, each invocation of π completes before the next one has to begin. The resulting SMR protocol is a generalization of the simple rotating leaders protocol in Section 2.3, with the (non-fault-tolerant) step of taking the leader's messages at face value with a (fault-tolerant) Byzantine broadcast computation.
The reduction not only produces an SMR protocol—it also extends the safety and liveness guarantees of the Byzantine broadcast subroutine to the resulting SMR protocol (with agreement and validity of the former implying consistency and liveness of the latter, respectively).

Theorem 2.6.1 SMR Reduces to BB

Under the stated assumptions, the SMR protocol produced by the reduction above satisfies consistency and liveness (with the same upper bound f on the number of Byzantine nodes).

Proof: For consistency, we can argue that all the honest nodes proceed in lockstep, with each appending the exact same ordered list of transactions in each iteration of the protocol. (They all start with the empty local history and, by induction, would then remain perfectly in sync forevermore.) Fix an arbitrary iteration of the SMR protocol. Because the Byzantine broadcast subroutine satisfies agreement, its invocation in this iteration terminates within T time steps (by assumption) and, whether or not the leader of iteration is Byzantine, with all honest nodes outputting the same list L. Thus all honest nodes do indeed append to their local histories the exact same list in each iteration.
For liveness, we need to slightly modify the statement of goal 2 in Section 2.1: every transaction submitted to

at least one honest node is eventually added to the local history of every honest node. (The protocol can't force Byzantine nodes to add anything to their local histories, nor can it force them to report transactions that they may have heard about.) So consider a transaction tx that a client sends to some honest node i. Because every node acts at the leader of an iteration infinitely often, node i will eventually be the leader of some subsequent iteration. In that iteration, if tx has not already been added to honest nodes' local histories, node i will include it in its list L^* of not-yet-executed transactions that it knows about. Because the Byzantine broadcast subroutine satisfies validity and because the leader/sender i is honest (with private input L^*), the subroutine terminates in at most T time steps with all honest nodes outputting L^*. All the honest nodes then append this list (and, in particular, the transaction tx) to their local histories. 😊

So, to produce a fault-tolerant SMR protocol, "all" we need to do is come up with a fault-tolerant Byzantine broadcast protocol. But wait, how do we do that?

2.7 Intuition: The $f = 1$ Case

The impatient reader can skip straight to Section 10 to learn about the Dolev-Strong protocol, which is a highly fault-tolerant solution to the Byzantine broadcast problem (and hence leads immediately to a fault-tolerant SMR protocol, via our rotating leaders reduction). But remember that one of the main points of this lecture is to build up our muscles for designing, analyzing, and having good intuition about consensus protocols. In that spirit, let's first think through how to use "cross-checking" to at least solve the Byzantine broadcast problem in the $f = 1$ case; in the equally instructive Section 2.8, we'll see why our protocol doesn't work in the $f = 2$ case and why further rounds of cross-checking are necessary.

We already know a simple protocol that solves the Byzantine broadcast problem in the $f = 0$ case (the sender broadcasts their private input, non-senders output whatever they heard from the sender) and that it doesn't work in the $f = 1$ case (a Byzantine sender could send different messages to different non-senders, leading to disagreeing outputs). Intuitively, honest non-senders should compare notes to check if they received consistence messages from the sender. A wrinkle in this idea is that there may be a Byzantine non-sender who tries to frame an honest sender by lying during the cross-checking phase. Given that we're working under the PKI assumption (Section 2.2.2), here's maybe the simplest way to implement the idea of "cross-checking" the messages sent out by the sender:

Note:-

A Simple Cross-Checking Protocol for Byzantine Broadcast

1. In the first time step, the sender sends its private value v^* to all the non-senders (along with its digital signature).
2. In the second time step, every non-sender i echoes the message m_i it received from the sender in the previous time step to all other non-senders, adding its own signature to m_i.
3. In the third and final time step, each non-sender chooses the most frequently referenced value in the messages it received from the sender and other non-senders (breaking ties in some consistent way, such as lexicographically). (The sender can simply output its private input v^*.

Honest nodes can easily ignore messages that could only have come from a Byzantine node— the hard part is dealing with plausibly deniable Byzantine behavior that, from the perspective of any single other node, could also in some universe reflect honest behavior. For example, an honest node can ignore any message received from the sender outside the first time step, and any message received from a non-sender outside the second time step. If an honest node doesn't receive a message when it's expecting one (e.g., due to a Byzantine sender who remains silent) or receives multiple messages, it can carry on as if it received a message with some canonical value, denoted $\perp$ (e.g., the empty list of transactions). This simple protocol is robust enough to withstand misbehavior by a single Byzantine node.

Proposition 2.7.1

When $f = 1, n \geqslant 4$, the simple cross-checking protocol above satisfies termination, agreement, and validity.

Proof: The protocol obviously terminates, after three time steps. To argue validity, assume that the sender is honest (otherwise, validity holds vacuously). The sender obviously outputs v^* , its private value. Each honest non-sender receives one vote for v^* signed by the (honest) sender (in the first time step) and at least one vote for v^* echoed and signed by another honest non-sender (in the second time step). (Because $n \geqslant 4$ and $f = 1$, there is at least one other honest non-sender.) An honest non-sender can receive at most one vote for a value other than v^* (from a Byzantine non-sender), so its majority vote computation in the third step will output v^*. Moving on to agreement, we only have to worry about the case of a Byzantine sender. (The validity argument above implies agreement in the case of an honest sender.) Because $f = 1$, in this case, every non-sender must be honest and will therefore echo the message received from the sender to all other non-senders in the second time step. Thus, at the start of the third time step, all the non-senders have received exactly the same information, namely the pool of all the messages sent out by the sender in the first time step. All nonsender therefore carry out exactly the same majority vote computation and hence compute the same final output (using here that in the event of a tie, all nodes tie-break in the same way). 😊

The simple cross-checking protocol does not solve the Byzantine broadcast problem when $f = 2$, however. Do you see why?

2.8 A Bad Example with $f = 2$

A protocol robust to Byzantine faults must succeed for every possible set of strategies that could be employed by Byzantine nodes, including "collusion" by those nodes (meaning coordinated deviations from the intended protocol). In effect, the Byzantine nodes may as well have secret and instantaneous communication channels amongst themselves. The next example should make clear the power of such conspiracies among the Byzantine nodes. Consider the simple cross-checking protocol of Section 2.7. Assume that the number n of nodes is even at least 4. Assume that $f = 2$, with a Byzantine sender and one Byzantine non-sender. We claim that there is a coordinated strategy for the two Byzantine nodes such that the protocol fails to satisfy agreement (see Figure 1):

- In the first time step, the (Byzantine) sender sends a "0" (along with its signature) to half of the honest non-senders (the set A in Figure 1) and a "1" (along with its signature) to the other half (the set B). (The argument in the previous section shows that this step along is not sufficient to break the protocol.)
- (The conspiracy.) Still in the first time step, the Byzantine sender sends two messages to the Byzantine non-sender, one with a "0" (and its signature) and the other with a "1". (Alternatively, the Byzantine sender can send its private key to the Byzantine non-sender, who can then create these two messages itself.)
- In the second time step, the Byzantine non-sender echoes the signed message with a "0" to the nodes in A and the one with a "1" to the nodes in B.

Thus, each of the Byzantine nodes uses the canonical ploy of sending conflicting messages to different honest nodes; the second step above defeats the PKI assumption and enables the Byzantine non-sender to use such a strategy. What do the honest non-senders output in the third time step? In the second time step, each node of A will hear $\frac{n}{2}$ votes for "0" (one from the sender, $\frac{n}{2} - 2$ from the other nodes of A, and the tie-breaking vote from the Byzantine non-sender) and $(\frac{n}{2}) - 1$ votes for "1" (from the nodes of B). Each such node will therefore output "0." Similarly, each node of B will output "1," violating agreement.
Three takeaways from this bad example:

1. Many seemingly good protocols are not robust to Byzantine faults, in large part because Byzantine nodes effectively have the power to fully coordinate their deviations from the intended protocol.
2. Even when a protocol is robust to Byzantine faults, the rich space of coordinated Byzantine strategies can make it difficult to rigorously prove it.
3. It would seem with more than one Byzantine node, more that one round of crosschecking is necessary. This is exactly what happens in the Dolev-Strong protocol in the next section, in which every additional round of cross-checking enables robustness to one additional Byzantine node.

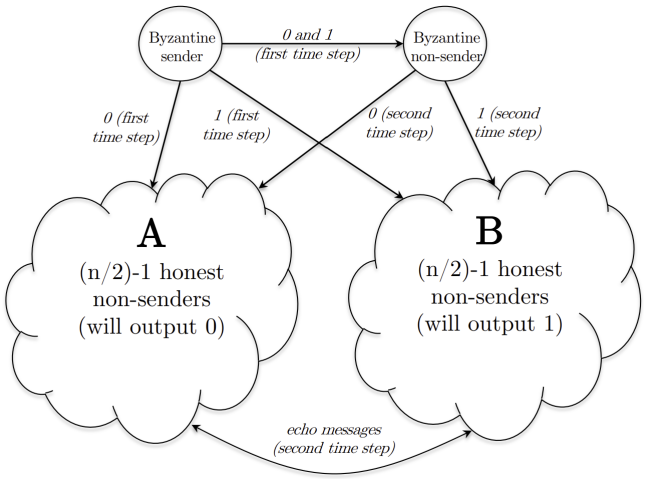


Figure 2.1: Illustration of a coordinated Byzantine strategy with $f = 2$. Whenever a node sends or echoes a message, it adds its own signature.

2.9 The Dolev-Strong Protocol

This section describes a classic (from 1983) solution to the Byzantine broadcast problem (in the permissioned and synchronous setting, and assuming PKI), due to Dolev and Strong [2].20 Coupled with the reduction in Section 2.6, this protocol gives a solution to the state machine replication problem (under the same assumptions).

2.9.1 Motivation

Full disclosure: you're not going to see the Dolev-Strong protocol mentioned very frequently in blockchain whitepapers and discussions. One reason for this is the protocol's heavy reliance on the synchronous model, which is an overly simplistic model of a global communication network like the Internet. A second issue is that the protocol always requires a number of time steps linear in f (the maximum-possible number of Byzantine nodes), which is more than one would like.
Nevertheless, there are several reasons to spend some quality time with the Dolev-Strong protocol:

1. It's one of the greatest hits of distributed computing. Just as it's satisfying to experience first-hand famous works of art, so too with famous algorithms and proofs in computer science.

2. It doesn't take that long. The protocol description is short, and the proofs of agreement and validity are clever but also short.

3. It would be pedagogically unsound to jump straight into the relatively complicated consensus protocols that form the basis of actual real-world blockchain protocols. Better to gradually ramp up the difficulty of the setting and the complexity of solutions. After spending some time getting comfortable in the relatively safe confines of the synchronous model, we can climb the next mountaintop and design consensus protocols that work well even under much weaker assumptions about the communication network.

2.9.2 Convincing Messages

We need one definition before proceeding to the description of the Dolev-Strong protocol. To motivate it, recall that in the simple cross-checking protocol in Section 2.7, non-senders compare notes in an attempt to pool together all the messages that the (possibly Byzantine) sender sent out in the first step—if any two of the messages sent differ, then the nonsenders can safely conclude that the sender is Byzantine, stop worrying about the validity requirement, and achieve agreement by all outputting some canonical value (e.g., the empty list of transactions). (And as we saw in Section 2.9, such pooling can be tricky to pull off if there are Byzantine non-senders acting in cahoots with a Byzantine sender.) The next definition establishes conditions under which an honest non-sender

can accurately conclude that a particular message was indeed sent by the sender to some node in the first time step. (For convenience, for the rest of this lecture we number the time steps starting from 0.)

Definition 2.9.1: Convincing Messages

A node i is convinced of value v at time step t if it receives a message prior to that time step that:

- references the value v;
- is signed first by the sender;
- is signed also by at least $t-1$ other distinct nodes, none of which are i. For example, if node 7 receives at time step 3 a message with a vote for "0" that is signed first by the sender (node 17, say) and also by nodes 23 and 29, then node 7 is convinced of the value 0 by this message.

2.9.3 Protocol Description

Here, finally, is the description of the Dolev-Strong protocol (i.e., the instructions carried out by honest nodes):

Note:-

The Dolev-Strong Protocol

Time step 0: the sender sends its private input v^* (with its signature attached) to all the non-senders, and outputs v^*.
Time step $t = 1, 2, \cdots, f+1$**:** if a non-sender i is convinced of a value v by a message m received prior to this time step and had not previously been convinced of v, the node adds its own signature to m and sends the resulting signed message (m, s) to all other non-senders.
Final output: for each non-sender, if it is convinced of exactly one value v, it outputs v; otherwise (having detected a Byzantine sender), it outputs $\perp$.

The symbol $\perp$ denotes some canonical value, such as the empty list of transactions. As usual, f denotes the maximum number of nodes that might be Byzantine; recall that its value is known up front, and indeed the protocol description replies on that knowledge. The protocol obviously makes use of the PKI assumption (so that nodes can verify each other's signatures), and our analysis of it in Section 2.10 will depend crucially on the reliable communication promised by the synchronous model.
Intuitively, time steps after the first represent the multiple rounds of cross-checking, with each (honest) non-sender telling the world about any values that its newly convinced of. Each non-sender is trying to catch a possible Byzantine sender red-handed by looking for contradictory messages sent out by the sender at time step 0.
Don't let the brevity of the protocol description fool you, it's really quite clever. This should be come clear to you in the next section, where we'll see short and sweet proofs that the protocol satisfies both validity and agreement (in the synchronous model), no matter how many of the nodes are Byzantine.

2.10 Properties of the Dolev-Strong Protocol

This section proves that, under assumptions 1–3 of Section 2.2, the Dolev-Strong protocol is a solution to the Byzantine broadcast problem: it satisfies termination, validity, and agreement. Termination is obvious from the protocol description; let's see why the other two properties hold, as well.

Theorem 2.10.1 Validity of the Dolev-Strong Protocol

Under assumptions 1–3 of Section 2.2, the Dolev-Strong protocol satisfies validity.

Proof: Assume that the sender is honest (as otherwise validity holds vacuously), with private value v^*. The sender thus follows the protocol, sending out signed copies of v^* to all the non-senders at time step 0 and then terminating immediately. Because signatures can't be forged (by our standing ideal signatures assumption) and no one other than the sender knows the sender's private key, there will never be any other messages that include the sender's signature. Looking at the second criterion of convincing messages

(Definition 2.9.1), we can conclude that no honest non-sender will ever be convinced of any value other than v^*. Are we done? Not quite. The worry is that some honest non-senders might get convinced of nothing (and thus output $\perp$, rather than v^*). This worried is unfounded. Because the sender is honest, it sends out signed copies of v^* to all non-senders at time step 0. Because we're working in the synchronous model, all of these messages will be received by their recipients before the start of time step $t = 1$. These messages satisfy Definition 10.1— because $t = 1$, no signatures other the sender's are required—and so all honest non-senders are convinced of v^* already in time step $t = 1$. The only messages in circulation for the entire protocol are the ones signed by the sender, all of which reference to the same value v^*. So, it's impossible to convince an honest node by any other value than v^* because the sender never sent any message with another value and noone else can forge a message from the sender with that value. 😄

Theorem 2.10.2 Agreement of the Dolev-Strong Protocol

Under assumptions 1–3 of Section 2.2, the Dolev-Strong protocol satisfies agreement.

Proof: Assume that the sender is Byzantine (validity already implies agreement for the special case of an honest sender). The plan is to prove that, when the protocol terminates, all honest non-senders have been convinced of exactly the same set of values. If true, this would imply agreement: the honest non-senders are either all convinced of the same single value v (in which case they all output v), the same set of 2 or more values (in which case they all output), or of no values at all (ditto). So, suppose that an honest non-sender i gets newly convinced of a value v by a message m received before the start of a time step t. We need to show that all other honest non-senders also get convinced of v before the end of the protocol.

Case 1: $t \leqslant f$. In this case, the timer has not yet gone off and i still has time to tell its colleagues about v. Precisely, because i is honest, it follows the protocol and adds its own signature to m and sends the resulting signed message (m, s) to all other non-senders. Because we're working in the synchronous model, every other honest non-sender j receive this message prior to the start of time step $t + 1$. The signed message (m, s) is signed first by the sender (because m was signed first by the sender) and also by at least t other distinct nodes (because m was signed by at least $t - 1$ other distinct nodes, none of which were i). If (m, s) includes j's signature, then j must have been convinced of v at some earlier point in time (honest non-senders only add their signatures to newly convincing messages); otherwise, because (m, s) satisfies the criteria of Definition 2.9.1, j becomes convinced of v at time step $t+1$.

Case 2: $t = f + 1$. In this case, i becomes convinced of v only as the game clock expires. There is no time for i to notify its honest non-sender colleagues about its new conviction, so the only hope is that i is late to the party and that everyone else independently became convinced of v (perhaps at time step $f + 1$, or perhaps earlier). Next we'll see that the Dolev-Strong protocol uses as many rounds as it does exactly so that this hope is in fact true.
For node i to become convinced of v for the first time at time step $f + 1$, according to Definition 2.9.1, it must have received a message m with v and signatures from at least $f + 1$ different nodes (one from the sender and at least $t - 1 = f$ from non-senders). Because at most f of the nodes are Byzantine (by assumption), at least one of these signatures must have been contributed by an honest (non-sender) node (node j, say). Because i received m prior to the start of time step $f + 1$, j contributed its signature at some earlier time step $t' \leqslant f$. (If you think about it, t' must equal f.) Whenever an honest non-sender adds a signature to a message, it broadcasts it to all other non-senders.The argument in Case 1 now applies (with j playing the role of i)-i didn't have time to notify all the other honest non-senders, but j did. Thus, every honest non-sender is convinced of v by time step $t' + 1 \leqslant f + 1$. 😄

Question 1

A good homework exercise is to convince yourself that Theorem 2.10.2 no longer holds if you stop the protocol one round early, after time step f rather than time step $f + 1$. (What would be the strategy for the Byzantine nodes?)

2.11 Discussion: How Big Can f Be?

Let's conclude by observing a very unusual property of the Dolev-Strong protocol with respect to the assumed upper bound f on the number of Byzantine nodes. As we noted, the protocol description depends on f, and the protocol's running time depends linearly on f. Maybe this isn't surprising—the more Byzantine nodes, the more challenging the problem and the harder we expect to work. But does the protocol ever stop being correct? Re-reading the proofs of Theorems 2.10.1 and 2.10.2, we see that both work no matter what f is. This is very unusual in distributing computing, and we won't see another result like it. Much more commonly, protocols become incorrect (and sometimes consensus problems even become unsolvable) once f crosses a certain threshold, such as $\frac{n}{3}$ or $\frac{n}{2}$.

Back to SMR. We should remember the reason we studied the Byzantine broadcast problem: protocols that solve it can be used as a subroutine (along with rotating leaders) to solve the problem that we really care about, state machine replication (Section 2.6). Combining the properties of this reduction (Theorem 2.6.1) with the guarantees of the Dolev-Strong protocol (Theorems 2.10.1 and 2.10.2) shows that the resulting SMR protocol satisfies consistency and liveness, no matter what f is. Many applications of SMR only make sense, however, when there's an honest majority (i.e., when $f \leqslant n/2$). Imagine each node is maintaining a copy of a database, or a copy of a blockchain. Think of a client who want to run a database query or check the current cryptocurrency balance of an account. By consistency, all honest nodes will respond to such a query with the exact same (correct) answer. Byzantine nodes might well lie and respond to such a query arbitrarily. If a strict majority of the nodes are honest, a user can send their query to all the nodes and take a majority vote of their answers to determine the correct one. If there's a 50/50 split of honest and Byzantine nodes, with the latter coordinating on a fabricated alternative state of the database or blockchain, a client cannot know whom to believe.

Looking ahead. Nodes running the Dolev-Strong protocol are contributing and verifying digital signatures all over the place, crucially relying on the PKI assumption. Could there be a different protocol for Byzantine broadcast that is equally fault-tolerant but does not require in-advance distribution of nodes' public keys? Next lecture is our first impossibility result: a "hexagon proof" that shows that the answer is "no." The existence of public-key cryptography and the ability to carry out a trusted setup really matter!

Chapter 3

Simulation, Indistinguishability, and the Necessity of PKI

3.1 On Impossibility Results

Recap and context. Last lecture, we proved a possibility result showing that, under a list of assumptions, there is a state machine replication (SMR) protocol satisfying the two properties that we care about, consistency and liveness. We did this by first reducing the multi-shot SMR problem to the single-shot Byzantine broadcast problem (i.e., showing how to build a good protocol for the former from one for the latter), and then presenting the Dolev-Strong protocol, which guarantees termination, agreement, and validity the Byzantine broadcast problem with any number of Byzantine nodes.

Beginning with this lecture and continuing in Lectures 4–6, we'll switch our focus from possibility results to impossibility results (before returning to a possibility result for the Tendermint protocol in Lecture 7). That is, we'll identify sets of assumptions under which there are no protocols that satisfy all the properties that we want. This lecture revisits the Byzantine broadcast problem from Chapter 2 and proves the "FLM impossibility result," which states that, under assumptions incomparable to the ones we made for the Dolev-Strong protocol, there is no protocol that satisfies all of termination, agreement, and validity. There are several reasons to spend some quality time with this impossibility result:

1. It's part of the distributed computing canon, and has a super-cool proof.
2. It highlights the importance of cryptographic and trusted setup assumptions (like the PKI assumption from Chapter 2).
3. It introduces two recurring themes in proofs of impossibility results in distributed computing: the idea of an adversary who performs a simulation of one or more honest nodes, and the idea of indistinguishability (in which an honest node gets caught in a catch-22 and can't distinguish between different worlds in which different behavior would be expected).

Digression: The point of impossibility results. Theory is great for proving possibility results, as we saw last lecture with the Dolev-Strong protocol. But arguably, theory might be even more uniquely suited for *impossibility* results that delineate what computers, algorithms, and protocols cannot do. Thinking back on the history of computer science, perhaps the first academic computer science paper ever was Alan Turing's 1936 paper that introduced the Turing machine model of computation[5]. The main result of that paper was an impossibility result showing that computers will never be able to solve the halting problem. Thus impossibility results have been part of the fabric of computer science literally from day zero as an academic field. A somewhat more modern example would be the development of *NP*-completeness, which explains why certain computational problems appear unsolvable by efficient algorithms. Distributed computing is another part of computer science that is defined in part by its deep and illuminating impossibility results. A lot of the richness of distributed computing as an academic discipline lies in the subtle curvature of the frontier between what can and cannot be done (as a function of exactly which assumptions you make). Impossibility results seems depressing. What are they good for? The point of an impossibility result is definitely not for some academic in an ivory tower to scold someone that they can't or shouldn't attempt to solve some problem. No matter what theorems I prove

in these lectures, people are not going to keep building new blockchain protocols and putting them out into the world (as they should). Rather, an impossibility result teaches you why you can't always have everything that you want, and about the compromises that are going to be required when you tackle a problem. For example, if you compare the major blockchain consensus protocols (sometimes called "layer-1s"), you'll see that none are perfect, and each has its own disappointing weaknesses. You might then wonder: "Why shouldn't I, or some other super-smart person, design one protocol to rule them all, with all the good points and none of the bad points of other protocols?" An impossibility result can inform us that, alas, no matter how smart any of us might be, such a protocol does not exist. It also provides you with a lens through which to evaluate and compare different blockchain protocols and the different compromises they (inevitably) make—for example, learning to identify which protocols "favor safely over liveness" or vice versa (more on this in later lectures). Finally, an impossibility result can also indicate that you're working in too demanding a model, with too few assumptions. For example, the FLP impossibility result (covered in Lectures 4–5) rules out good consensus protocols in the extremely demanding "asynchronous model," and this result will guide us toward the definition of the "partially synchronous model" in Lecture 6. (Whereas without the FLP impossibility result, it's not clear anyone would have bothered to write down that model.) As we'll see, the partially synchronous model will be the perfect sweet spot between more extreme models—its assumptions are weak enough that it forces you to design protocols that really are useful in the real world, yet strong enough to escape the FLP impossibility result and allow for provable guarantees. That's the reason we are spending time on these impossibility results they tell us when trade-offs are absolutely unavoidable and then we can understand different consensus protocols as taking different approaches to these necessary trade-offs.

3.2 The FLM Impossibility Result

3.2.1 The Byzantine Broadcast Problem

Recall from Chapter 2 the definition of and goals for the Byzantine broadcast problem (and see that lecture for further discussion). There is an a set $\{1, 2, 3, \cdots, n\}$ of nodes—this is common knowledge to all when a protocol commences, as is the identity of a sender node. The sender has a private input $v^* \in V$. (For this lecture's impossibility result, we'll only need to use set $V = \{0, 1\}$.)
The goal is to design a protocol with three properties:

Note:-

Desired Properties of a Byzantine Broadcast Protocol

1. **Termination:** Every honest node i eventually halts with some output $v_i \in V$.
2. **Agreement:** All honest nodes halt with the same output (whether or not the sender is honest).
3. **Validity:** If the sender is an honest node, then the common output of the honest nodes is the private input v^* of the sender.

3.2.2 Statement of the Impossibility Result

The impossibility result in this lecture was first established by Pease, Shostak, and Lamport [3] (the same authors from the "Byzantine generals" paper mentioned last lecture, with their names in a different order). We'll present a later (super-slick) proof by Fischer, Lynch, and Merritt[1]. For the theorem statement, recall that we use f to denote a known upper bound on the maximum number of Byzantine nodes, where a Byzantine node can deviate from the intended protocol in arbitrary ways. (The bigger the f, the harder it is for honest nodes to not get confused and achieve consensus.)

Theorem 3.2.1

In the synchronous model with $f \geqslant \frac{n}{3}$, there is no Byzantine broadcast protocol that satisfies termination, agreement, and validity.

This is, Byzantine broadcast is possible in the synchronous model only if more than twothirds of the nodes are honest and correctly follow the protocol. Here "synchronous model" is the same model of communication

that we used for the Dolev-Strong protocol in the last lecture—all nodes share a global clock, and every message sent in a time step arrives at its recipient prior to the beginning of the next time step.

3.2.3 Cryptography and Trusted Setups Matter!

Looking at the statement of Theorem 3.2.1, you should have a question: Didn't we just (in Chapter 2) learn a protocol (the Dolev-Strong protocol) that does solve the Byzantine broadcast problem (in the synchronous model) no matter how many Byzantine nodes there are? Yes we did! But neither the proof from last lecture nor the forthcoming proof of Theorem 3.2.1 are incorrect. The reason there's no contradiction is that the two results apply under slightly different sets of assumptions (in particular, different cryptographic/trusted setup assumptions). So, as we go through the proof of Theorem 3.2.1 in the next section, your homework is keep an eye out for any steps in the proof that wouldn't apply to the Dolev-Strong protocol (and there must be some, since that protocol escapes the impossibility result). We'll discuss this point at length in Section 3.4.

3.2.4 Two Simplifying Assumptions

To keep the length of this lecture reasonable, let's make two simplifying assumptions (both of which can be removed with a bit more work). First, let's focus on deterministic protocols (with no random coin flips). Allowing randomization doesn't really save you from the impossibility result (see [2]), but let's not worry about it further. Second, let's focus on the simplest-possible case of the impossibility result, with $n = 3$ and $f = 1$ (three nodes, one of which might be Byzantine). Your first reaction might be that this assumption trivializes the result, but the truth is the exact opposite—this special case already captures all of the complexity and nuance of the general impossibility result. A good homework problem is to extend this lecture's argument for the special case of $n = 3$ and $f = 1$ to a proof of Theorem 3.2.1 in its full glory. One approach is to use a reduction—that is, to show how to use a protocol that allegedly solves the Byzantine broadcast problem for some n and $f \geqslant \frac{n}{3}$ to build a different protocol that solves the $n = 3$ and $f = 1$ case (which would be a contradiction). Alternatively, the proof in the next section can be reworked to directly apply to your favorite choice of n and $f \geqslant \frac{n}{3}$.

3.2.5 Some Vague Intuition

Before getting to the formal proof of Theorem 3.2.1, let me give you some intuition about why the result might be true, and what's fundamentally driving the magical threshold of $\frac{n}{3}$. (By the way, if you skip the proof, don't forget to read Section 3.4 for a discussion of why the Dolev-Strong protocol doesn't contradict Theorem 3.2.1.) Warning: the intuition will be somewhat vague. The super-slick proof in Section 3.3 more or less encodes this intuition, though there is some distance between them.

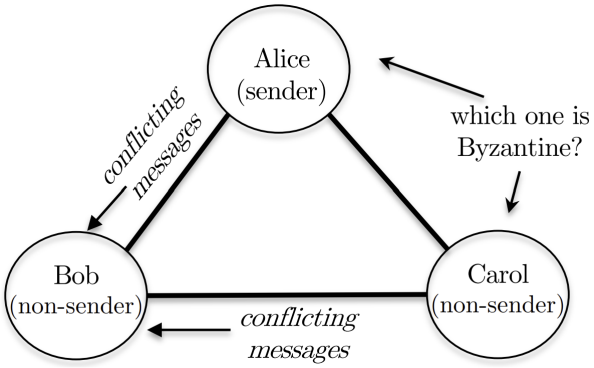


Figure 3.1: Vague intuition for Theorem 3.2.1. Bob can't tell if Alice (the sender) is Byzantine and Carol is honestly echoing her messages or if Carol is Byzantine and trying to frame Alice.

Think of a set of three nodes, call them Alice, Bob, and Carol, shown as vertices in Figure 3.1. The edges in the figure indicate that each of three nodes knows about and can communicate with the other two nodes. Assume that Alice is the designated sender. The basic idea is that one node may have enough information to deduce that

one of the other two nodes is Byzantine but not enough to deduce which one it is. For example, assume the role of Bob. He might well hear conflicting information from Alice (the sender) and Carol (the other non-sender), for example if Carol is supposed to repeat to Bob any messages she received from Alice. One plausible scenario is that Carol is honestly echoing the messages she heard from Alice, but Alice is Byzantine and sending conflicting messages to Bob. But it's also totally possible that Alice is honest and sent out consistent information, but Carol is Byzantine and attempting to frame Alice. Unable to determine who is to blame, Bob doesn't know what he should output. (Whereas if $n = 4$ and $f = 1$, Bob could plausibly figure out the right output by resorting to some type of majority vote over the other three nodes.) If you're really on top of things, you might be suspicious of this informal argument (in particular, of Carol's ability to frame Alice) and have an inkling about why an argument of this type might not apply to the Dolev-Strong protocol. In any case, spot the inevitable step(s) of the proof in the next section that doesn't apply to the Dolev-Strong protocol!

3.3 The FLM Proof of Theorem 3.2.1

3.3.1 Preamble (Optional)

The FLM proof of Theorem 3.2.1 is impressively clever—almost like they already knew the result was true and wanted to back out the slickest proof possible. (Actually, maybe that's exactly what they were doing—recall that the theorem was proved earlier in [3].) No one who sees this proof thinks it would be easy to come up with, and it's normal to not grok it the first time you see it. On a first reading, your main goal should be convince yourself that it's correct. With some repeated readings and rumination, you'll be able to internalize it.

The challenge of impossibility results. The main reason that impossibility results in computer science are so challenging is the richness of the design space—you need to rule out any possible solution, no matter how creative or crazy. This is exactly what made Turing's impossibility result [5] such a breakthrough—no algorithm, present or future, no matter how wild, could ever solve the halting problem. For another example, you might have heard about the P 6= NP conjecture, which roughly asserts that there are no algorithms for any NP-hard problems (satisfiability, traveling salesman problem, etc.) that are guaranteed to improve substantially over (exponential-time) exhaustive search (see e.g. [4] for details). And the whole reason nobody has been able to prove the conjecture yet is the richness of the algorithm design space. With so many crazy possibilities for polynomial-time algorithms— ever seen Strassen's matrix multiplication algorithm?)—how could one ever rule all of them out? The exact same issue can make impossibility results for distributed protocols difficult. For example, suppose we try to prove Theorem 3.2.1 by contradiction. That is, we assume that the theorem is false and that there in fact a protocol (call it π) for the Byzantine broadcast problem with $n = 3$ and $f = 1$ guaranteed to satisfy termination, agreement, and validity. If we can derive a contradiction (showing that our initial assumption is false), we'll have completed the proof. Here's the thing, though: we have no idea what π looks like, as it could be arbitrarily crazy event-driven piece of code. One sort of trivial thing we can say about π is that it is designed to work in the $n = 3$ case and we're in the permissioned setting so that means if you're a node running π basically you're going to know about two IP addresses different than yours and these are the two nodes you should be expecting to hear messages from you're not expecting your messages from anyone other than these two IP addresses and you should also feel free to send messages to those nodes if you want. How can we write down a single argument that somehow works simultaneously for all of the infinitely possible π's?

Simulation and indistinguishability. There is one thing we can do in the proof that uses the assumed protocol π in a generic way: run it! That is, we (or an adversary) can imagine deploying it on one or more nodes and then seeing what happens. (Proofs in theoretical computer science often use algorithms in this "black-box" way. E.g., reread any NP-completeness proof that you might have seen in the past.) This idea of "simulation"—in our case, of hypothetical honest nodes by a devious Byzantine node—is one of the two key themes to look for in the FLM proof of Theorem 3.2.1.

The second theme of the proof is "indistinguishability," which we alluded to in the vague intuition given in Section 3.2.5. Indistinguishability refers to a catch-22 situation faced by an honest node, in which the node does not have enough information to figure out which of two plausible scenarios is the actual reality.

Example 3.3.1 (The example for this case is when we had alice, bob and carol we were thinking of alice as the sender we were sort of you know playing the role of bob an honest non-sender and we talked about how you know on the one hand alice might be byzantine send out conflicting information on the other hand carol might be the byzantine one and might try to frame alice and convince bob that alice is byzantine when in fact it's carol the byzantine one so you know what this is trying to say is that you know bob is now in this sort of catch-22 sort of impossible situation.It really sort of based on what it's seen, based on the messages received can't distinguish between whether alice is the byzantine one or whether it's carol and conceivably bob's appropriate behavior with respect to validity and agreement is going to depend on which of those two situations is the case and this is what makes consensus hard; honest nodes find themselves in this catch-22, they might be in world A, they might be in world B and they can't distinguish which one they're in and if the sort of requirements demand different behavior of that node in the two different settings the node's stuck and he can only do one thing it's going to be right in one of the worlds and it's going to be wrong in the other world.)

If validity and/or agreement demand different outputs in the two cases, then the node is hopelessly stuck (whatever it outputs, it will be an incorrect output for one of the plausible scenarios). With a specific protocol, we can start imagine ways in which a Byzantine node might trap an honest node into a catch-22 situation (along the lines of Section 3.2.5). But how can we do it in a generic way, that applies no matter what π is?

3.3.2 The Hexagon (a Thought Experiment)

Next is the really, really ingenuous step of the Fischer-Lynch-Merritt proof, which involves a thought experiment carried out on a 6-cycle of nodes (a "hexagon"). (Once you've convinced yourself that this thought experiment makes sense, the rest of the proof won't be too bad.) The basic idea is to effectively make two copies of each of Alice, Bob, and Carol, in order to force them to participate simultaneously in multiple worlds that demand different things of them. Accordingly, we're going to buy six machines and deploy an allegedly correct Byzantine broadcast protocol π on each of them.
To make sense of this idea, let's be clear on the inputs that the protocol π is expecting to find when it's first fired up on some node i (in a locally stored initialization file, if you like):

(a) the names and IP addresses of two other nodes (recall the protocol π is designed for the case of $n = 3$);

(b) among node i and the two other nodes, which one is the sender (there should be exactly one sender);

(c) if node i is the sender, its private input.

The designer of the protocol π surely had in mind a scenario in which the initialization files at i's two neighbors (call them j and k) list the nodes i, k and i, j, respectively—this would be the case in any bona fide instance of Byzantine broadcast with $n = 3$, and these are the only cases that π's designer is responsible for. However, at least in principle, we could run the protocol π at node i even if j's and k's initialization files did not satisfy this consistency property. We can't count on π satisfying any properties in this case; for example, the initialization file inconsistencies might well cause the protocol to run forever. But because π is just a piece of code and node i's initialization file has all the requisite ingredients, we can nonetheless press play on node i and see what happens. The hexagon thought experiment exploits the idea above, running six copies of π on machines with inconsistent initialization files (Figure 3.2, with edges indicating pairs of nodes that know about each other before the protocol starts):

Note:-

Thought Experiment Instructions

1. Buy a node A and set it up with the following initialization file:
   (i) A's neighbors are the nodes B and C (specified by name and IP address);
   (ii) A is a sender while B, C are non-senders;
   (iii) A's private input is 0.

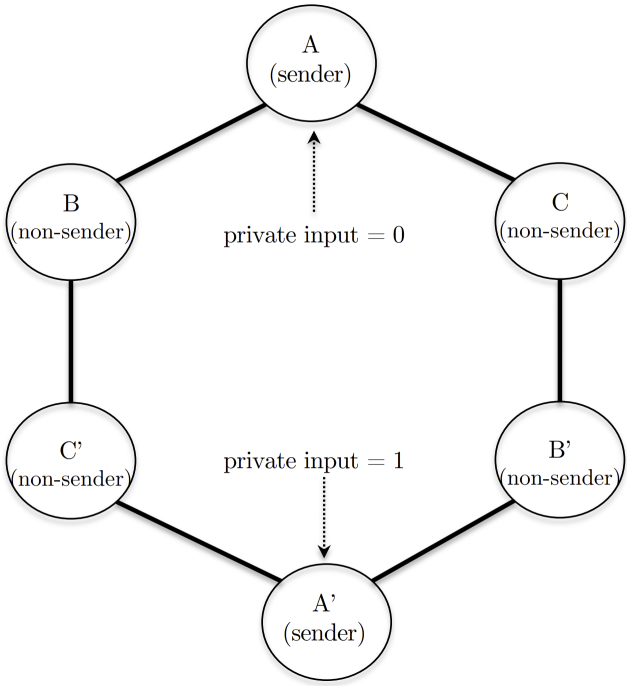


Figure 3.2: The hexagon thought experiment. Each machine runs the protocol π with the indicated two neighbors (and, if a sender, with the indicated private input).

2. Buy a node B and set it up with:
   (i) B's neighbors are the nodes A and C'
   (ii) A is a sender while B, C' are non-senders. (Think of C' as having the same name as C (e.g., each thinks it's the real node 7) but a different IP address.)
3. Buy and setup a node C with:
   (i) neighbors A and B'
   (ii) A is a sender while B' and C are non-senders
4. Buy and setup a node C' with:
   (i) neighbors A' and B;
   (ii) A' is a sender while B, C' are non-senders.
5. Buy and setup a node B' with:
   (i) neighbors A' and C;
   (ii) A' is a sender while B', C are non-senders.
6. Buy and setup a node A' with:
   (i) neighbors B' and C'
   (ii) A' is a sender while B', C' are non-senders;
   (iii) the private input of A' is 1.

This whole endeavor should seem bizarre. The only point to understand now is that, given the code of a protocol π, no one can stop you from buying six machines, installing π on each of them, and setting up their initialization files as above. Because each initialization file has exactly the information (a)–(c) expected by π—for

all the node knows, it's participating in a bona fide instance of Byzantine broadcast with $n = 3$—no one can stop you from pressing play simultaneously on all six machines. All bets are off about what happens next (π was not designed with this experiment in mind), but no one can stop you from running the experiment to see what the nodes' eventual outputs will be.

3.3.3 Proof Strategy

Because the initialization file of each node in the hexagon thought experiment typechecks with the expectations of the protocol π, you can run the experiment of running π simultaneously on all six nodes. But the overall setting in the thought experiment ($n = 6$, not all nodes no about all other nodes) certainly doesn't typecheck with the one π is designed for ($n = 3$, set of nodes common knowledge). Thus any properties that π might have in the latter scenario (e.g., validity) need not carry over to the former scenario (e.g., with two senders with different private inputs, it's not even clear what validity should mean). This point has important consequences for our proof strategy.

In our proof, to derive a contradiction, presumably we are going to use the assumptions that π satisfies termination, agreement, and validity. (Achieving any two of the three properties is easy, so there's no way to prove an impossibility result without using all three assumptions.) How are we going to use the assumed properties of π in the proof, given that we can't appeal to them directly in the hexagon thought experiment?

To build a bridge between the hexagon experiment (where π has no guarantees) and bona fide three-node instances of Byzantine broadcast (where π satisfies termination, agreement, and validity), we're going to show that the hexagon effectively encodes three different three node Byzantine broadcast instances. In each of these instances, agreement or validity will impose a constraint on the outputs of a pair of nodes, and these constraints will carry over to the hexagon. We'll see that the three constraints are mutually incompatible, contradicting the fact that the hexagon thought experiment must have some well-defined output. In our example, all 6 nodes eventually output 0 or 1(π satisfies termination).

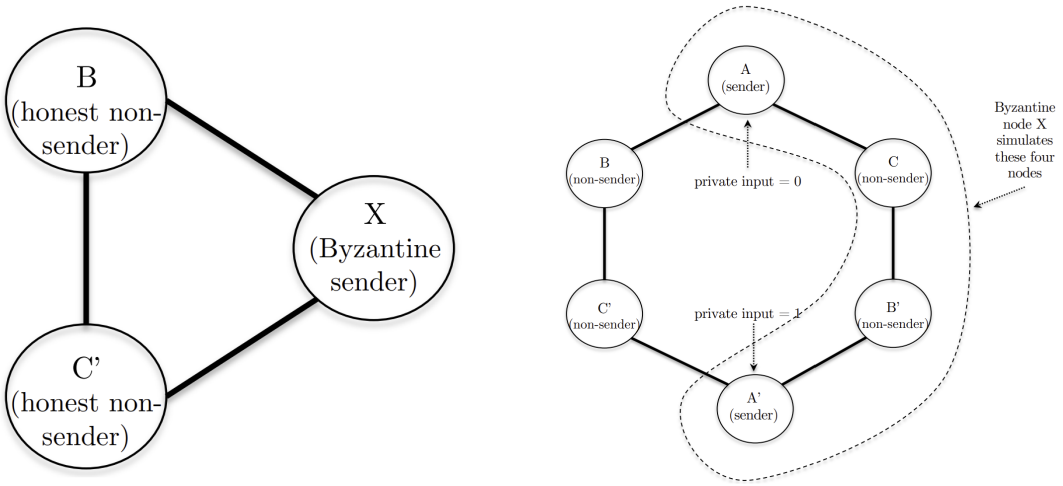
3.3.4 Byzantine Broadcast Instance 1

Let's see the first instance of Byzantine broadcast that is effectively "embedded"in the hexagon through experiment. Consider the instance shown in Figure 3.3(a), with a Byzantine sender (node X) and two honest non-senders (B and C'). As a bona fide instance of Byzantine broadcast (with $n = 3$ and one Byzantine node), we can appeal to the promised guarantees of the assumed protocol π, specifically its termination and agreement properties (validity is irrelevant because the sender is Byzantine). Remember what these properties say: no matter what a Byzantine node does, no matter how crazy its strategy, the honest nodes must eventually terminate, and upon termination output the same value. Intuitively, the protocol π must be robust to X sending out conflicting messages. But given that we know nothing about how π works, how can we know which messages might confused the nodes that are running π honestly? Could there be a generic "send conflicting messages" strategy for X that applies to every possible protocol π? Here's one crazy strategy that the Byzantine node X in Figure 3.3 could do: So this is where this will be the first appearance of the simulation idea in this proof. The adversary is going to simulate the protocol π actually it's going to simulate four copies of the protocol π. so why four copies? well it's going to be the four machines in the thought experiment, the four machines in the sixth cycle other than B and C'. So that's going to be the chain starting with machine A running as a sender with input zero it's going to include the non-senders C and B' and then the sender with input 1:

$$A \leftrightarrow C \leftrightarrow B' \leftrightarrow A'$$

in the hexagon thought experiment, in which A', C, B' and A' run π honestly with the initialization parameters specified in Section 3.3.2. (Remember, simulation is one of the two key themes of this proof.) Think of it this way: the Byzantine node X spins up four virtual machines on its node, each running a separate copy of π with the appropriate initialization parameters. The Byzantine node monitors the progress of each copy of π. Whenever π instructs the virtual machine masquerading as A to send a message to A's neighbor B (which is X's actual neighbor in the Byzantine broadcast instance), node X sends that message to B (over the communication network). Similarly, any messages X receives from B (over the communication network) are forwarded to the virtual machine on X that is running π as A. Messages between the virtual machine running as A' and the (actual) node C' are handled analogously, over the communication network. Finally, whenever one virtual machine wants to send a message to another one (e.g., messages between the virtual machines running as C and B'), the Byzantine node X can directly deliver that message to the destination machine, without ever touching

the communication network. In the end, the Byzantine node X interacts with its neighbor B as if it were the node A in the hexagon (running π honestly with private input 0) and with C' as if it were the node A' in the hexagon (running π honestly with private input 1). This strategy is well defined for any protocol π and thus serves as our generic way for a Byzantine node to send conflicting messages.



(a) Byzantine broadcast instance

(b) Hexagon thought experiment

Figure 3.3: Left: Byzantine broadcast instance 1 (with $n = 3$ and one Byzantine node). Right: by simulating four nodes in the hexagon experiment, the Byzantine node X can force B and C' to operate identically in the triangle and in the hexagon. The agreement property of π dictates that B and C' output the same value.

At this point you should agree that this "simulate four nodes on the hexagon" strategy is something that the Byzantine node X could do. But what's the point of this crazy strategy? The point is that it tricks the honest nodes B and C' to behave exactly as if they were in the hexagon experiment. That is, because the initialization parameters and the sequences of messages received by B and C' are exactly the same in the triangle in Figure 3.3(a) (by construction of X's strategy) and in the hexagon in Figure 3.3(b), they are kept in the dark cannot distinguish which is the actual reality and must operate identically in both cases. (Remember, indistinguishability is the other key theme of the proof.)

Now we can see how to translate π's assumed properties in bona fide Byzantine broadcast instances (like the triangle in Figure 3.3(a), with the above Byzantine node strategy) to properties of its behavior in the hexagon experiment. Specifically, in the triangle, because π satisfies termination and agreement, nodes B and C' eventually output something, and the outputs must be the same. (This statement is true no matter what strategy X uses.) Because B and C' operate identically in the triangle (with the specified strategy for X) and in the hexagon thought experiment, the outcome must be the same:

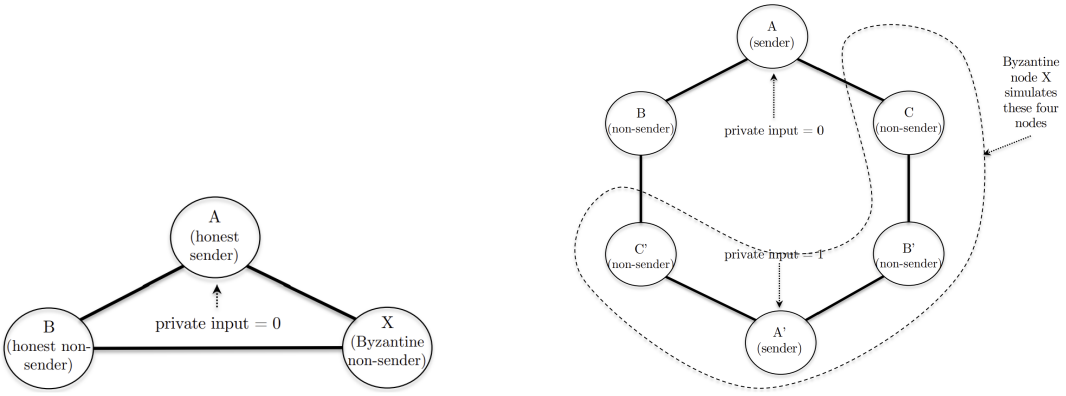
in the hexagon thought experiment, the nodes B and C' output the same value. (1)

Good news: If you followed the argument in this section, then you've completed all of the hard parts of understanding the FLM proof of Theorem 3.2.1. We need to talk through two more scenarios before arriving at a contradiction, but the pattern of those two arguments will be the same as this one.

3.3.5 Byzantine Broadcast Instance 2

The first scenario relied on the assumed agreement of the protocol π; the second and third ones will rely on validity. Validity applies only with an honest sender, so in these two scenarios the Byzantine node X will be a non-sender (see the triangle in Figure 3.4(a)).

The honest nodes A and B in the triangle play the same roles as their namesakes in the hexagon, and particular the honest sender A has a private input of 0. It's going to be the same as before, so we're going to look



(a) Byzantine broadcast instance

(b) Hexagon thought experiment

Figure 3.4: Left: Byzantine broadcast instance 2. Right: by simulating four nodes in the hexagon experiment, the Byzantine node X can force A and B to operate identically in the triangle and in the hexagon. The validity property of π dictates that A and B both output 0.

at the four nodes on the sixth cycle other than the two that are present, so in this case it's going to be the chain of nodes between C and C' and the adversary is just going to simulate four copies of the protocol π; again it has its computer, it has its four virtual machines; one of those is running C one is B', one is A' and one is C'. So the the byzantine node X will interact with the honest sender A as if it was the node C in the thought experiment and it will interact with the honest non-sender B as if it was the node C' in the thought experiment and then the adversary the the byzantine node X will also be simulating the other nodes of the six cycle A' and B' to ensure that it's faithfully replicating all of the computation and communication that would be happening in the thought experiment. To keep the nodes A and B in the dark as to whether they reside in the triangle or the hexagon, the Byzantine non-sender X in the triangle runs four copies of π (in separate virtual machines) to simulate the remaining four nodes of the hexagon (Figure 3.4(b)):

$$C \leftrightarrow B' \leftrightarrow A' \leftrightarrow C'$$

That is, the Byzantine node X interacts with the honest sender A as if it were the node C in the hexagon, and with the honest non-sender B as if it were the node C' in the hexagon (while also simulating the nodes B' and A' to keep track of which messages C and C' would be sending). Because π satisfies termination and validity in every bona fide three-node instance of Byzantine broadcast (like the triangle) with an arbitrary Byzantine node strategy (like X's simulation strategy), nodes A and B must eventually output 0 (i.e., A's private input) on the triangle. By virtue of operating identically in the hexagon experiment (by construction of X's strategy), those two nodes must also eventually output 0 in the hexagon:

$$\text{in the hexagon thought experiment, the nodes A and B output 0.} \quad (2)$$

Again, the seemingly bizarre simulation strategy by the Byzantine node X is what enables the transfer of constraints imposed on nodes' outputs in the triangle to those in the hexagon.

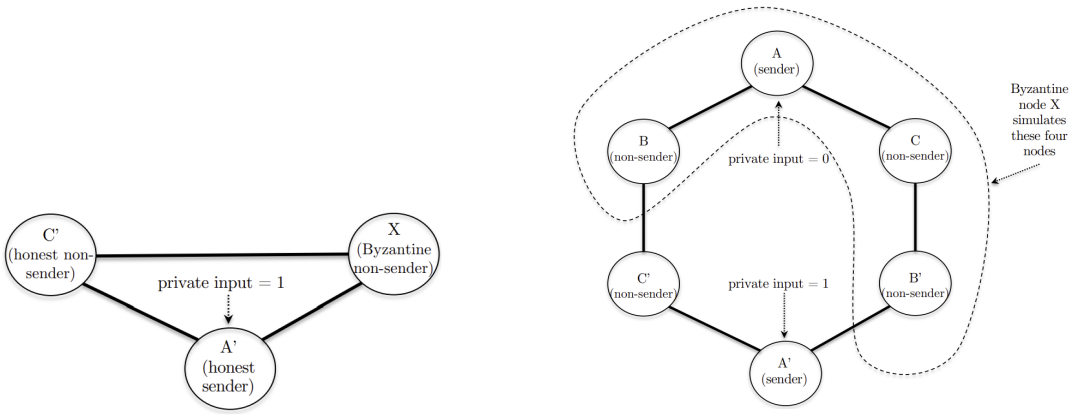
3.3.6 Byzantine Broadcast Instance 3

The third scenario is very similar to the second one, with an honest sender A' (this time with private input 1), an honest non-sender C', and a Byzantine non-sender X (see the triangle in Figure 3.4(a)).

To keep the honest nodes A' and C' in the dark as to whether they reside in the triangle or the hexagon, the Byzantine non-sender X runs four copies of π to simulate the remaining four nodes of the hexagon (Figure 3.4(b))

$$B \leftrightarrow A \leftrightarrow C \leftrightarrow B'$$

interacting with A0 as if it were B' and with C' as if it were B. Because π satisfies termination and validity in every bona fide three-node instance of Byzantine broadcast (like the triangle) with an arbitrary Byzantine node



(a) Byzantine broadcast instance

(b) Hexagon thought experiment

Figure 3.5: Left: Byzantine broadcast instance 3. Right: by simulating four nodes in the hexagon experiment, the Byzantine node X can force A' and C' to operate identically in the triangle and in the hexagon. The validity property of π dictates that A' and C' both output 1.

strategy (like X's simulation strategy), nodes A' and C' must eventually output 1 (i.e., the private input of the honest sender A') on the triangle. By virtue of operating identically in the hexagon experiment (by construction of X's strategy), those two nodes must also eventually output 1 in the hexagon:

in the hexagon thought experiment, the nodes A' and C' output 1. (3)

3.3.7 Completing the Contradiction

We looked at three different three-node Byzantine broadcast instances and (appealing to π's assumed guarantees) deduced constraints on the outputs of the honest nodes in each. We chose a simulation strategy for the Byzantine node in each of these instances to force the honest nodes to operate identically to how they would act in the hypothetical hexagon thought experiment defined in Section 3.3.2, thereby transferring the constraints from the three instances over to hexagon. These constraints (1)–(3) assert that, in the hexagon thought experiment, nodes B and C' must terminate and output the same thing; B must output 0; and C' must output 1. Whatever the outcome of the hexagon experiment might be (and it must be something), it can't possibly satisfy these three mutually inconsistent constraints. This completes the contradiction, implying that the assumed protocol π cannot exist. That is, there cannot be a Byzantine broadcast protocol with $n = 3$ and $f = 1$ that guarantees termination, agreement, and validity.

3.4 Discussion

The main result of Chapter 2 is that, in the synchronous model, you can solve the Byzantine broadcast problem (i.e., with a protocol that satisfies termination, agreement, and validity) with any number of Byzantine nodes. The main result of this lecture is that, in the synchronous model, you can't solve the Byzantine broadcast problem if at least one-third of the nodes can be Byzantine. What gives?

3.4.1 Resolving the Contradiction

No prizes for guessing the answer, which appears in the title of this lecture. In Chapter 2, we introduced the public key infrastructure (PKI) assumption as an extension of the usual permission assumption: not only do all the nodes know the names and IP addresses of all the nodes, but also their public keys. (Each node is assumed to have a distinct public key-private key pair, with the private key known only to the node and the public key known to all.) This is an example of a trusted setup assumption, asserting that a certain computation (in this case, public key distribution) is done correctly in advance of the protocol's commencement, remaining silent on how this computation might actually happen.

Our analysis of the Dolev-Strong protocol in Chapter 2 relied on three assumptions: the permissioned setting (with all node names common knowledge at the start of the protocol); synchronous setting (reliable message delivery); and the PKI assumption (so that nodes begin the protocol with the ability to verify each others' signatures). The proof in Section 3.3 does not violate the permissioned setting assumption—in all three of the Byzantine broadcast instances considered, the node set is known to all nodes up front. Neither does it violate the synchronous setting assumption—it relies only on devious communication strategies for Byzantine nodes and not on any manipulation of the timing of message deliveries. Thus, by the process of elimination, we can conclude that Theorem 3.2.1 cannot possibly remain true under the PKI assumption (if it did, it would contradict what we've already proved about the Dolev-Strong protocol). Given that the theorem is false with the PKI assumption, the proof in Section 3.3 must somehow violate that assumption. But now exactly would common knowledge of nodes' public keys break the proof?

3.4.2 Can We Salvage the Proof?

For the hexagon thought experiment in Section 3.3.2 to even make sense under the PKI assumption, we need to add the relevant cryptographic keys to nodes' initialization files. The new file format is:
(I1) the names, IP addresses, and public keys of two other nodes;
(I2) among node i and the two other nodes, which one is the sender;
(I3) if node i is the sender, its private input;
(I4) a public key-private key pair for node i (distinct from those of the other two nodes).

The most obvious way to modify the thought experiment would then be (see Figure 3.6):

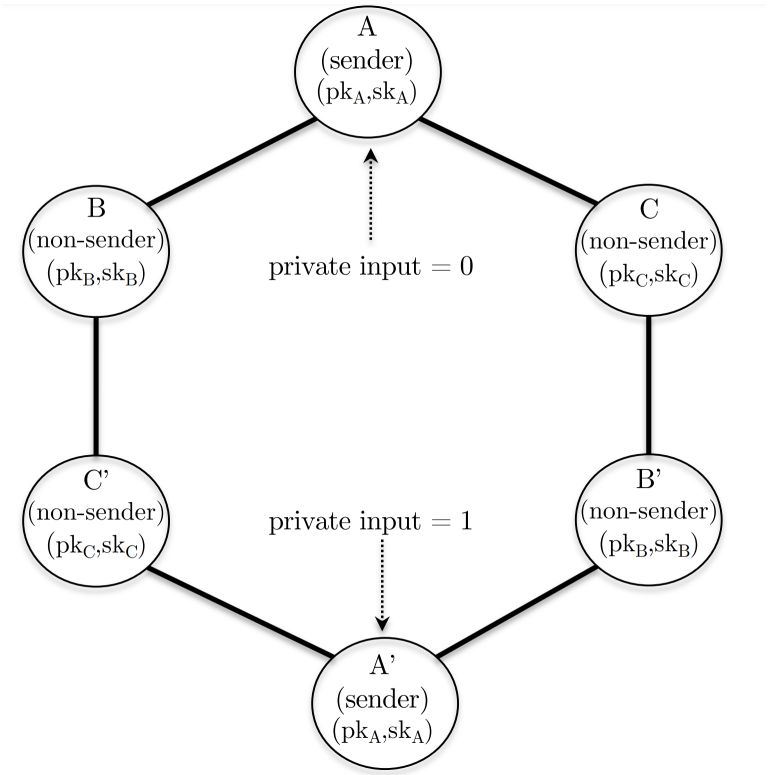


Figure 3.6: Extension of the hexagon thought experiment to incorporate the PKI assumption.

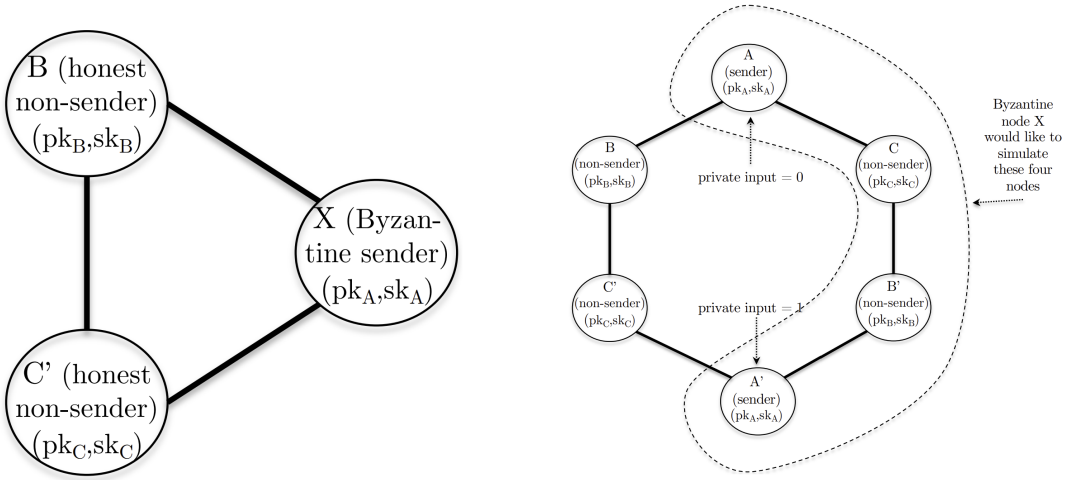
Note:-

Revised Thought Experiment

1. Setup node A with:
   (i) neighbors B and C (specified by name, IP address, and public key);
   (ii) A is a sender while B, C are non-senders;
   (iii) A's private input is 0.
   (iv) a public key-private key pair (pk_A, sk_A).
2. Buy a node B and set it up with:
   (i) B's neighbors are the nodes A and C'
   (ii) A is a sender while B, C' are non-senders.
   (iii) a public key-private key pair (pk_B, sk_B). (Think of C' as having the same name and key pair as C (e.g., each thinks it's the real node 7 and has the corresponding credentials) but a different IP address.)
3. Setup node C with:
   (i) neighbors A and B'
   (ii) A is a sender while B' and C are non-senders
   (iii) a public key-private key pair (pk_C, sk_C)
4. Setup node C' with:
   (i) neighbors A' and B;
   (ii) A' is a sender while B, C' are non-senders.
   (iii) a public key-private key pair (pk_C, sk_C)
5. Setup node B' with:
   (i) neighbors A' and C;
   (ii) A' is a sender while B', C are non-senders.
   (iii) a public key-private key pair (pk_B, sk_B).
6. Setup node A' with:
   (i) neighbors B' and C'
   (ii) A' is a sender while B', C' are non-senders;
   (iii) the private input of A' is 1.
   (iv) a public key-private key pair (pk_A, sk_A).

Now that we have a well-defined thought experiment with the PKI assumption, let's try to replicate the argument in Section 3.3. (The proof hasn't broken yet, but it has to break somewhere. . .) The next step is to consider a Byzantine broadcast instance analogous to the one in Section 4.4 (Figure 3.7(a)) and the strategy for the Byzantine node X in which it simulates the other four vertices in the revised hexagon experiment (Figure 3.7(b)). Remember that the point of this strategy is to keep the two honest nodes in the dark as to whether they reside in the triangle or the hexagon, in which case any constraints we can deduce for nodes' behavior in the triangle would carry over to the hexagon.

But is the Byzantine node X really in a position to pull off such a simulation? The assumed Byzantine broadcast protocol π might well instruct every node to sign every message it sends (as in the Dolev-Strong protocol). The adversary X knows its public key-private key pair (pk_A, sk_A) in the Byzantine broadcast instance, as well as the other two public keys pk_B and pk_C, but it does not know the other two private keys sk_B and sk_C. Thus, while the adversary is perfectly positioned to simulate the nodes A and A'—the only ones the honest nodes B and C' communication with directly—it cannot sign a message on behalf of nodes B' or C and is therefore unable to simulate them. (Remember our permanent assumptions that cryptography exists and that a signature for an as-yet-unseen message cannot be forged without knowledge of the appropriate private key.) In sum: our proof of Theorem 3.2.1 made crucial use of strategies in which a Byzantine node simulates multiple other honest nodes, and these strategies become impossible to carry out under the PKI assumption.



(a) Byzantine broadcast instance

(b) Hexagon thought experiment

Figure 3.7: An attempt to extend the argument of Figure 3.3 to the PKI setting. The simulation strategy could require the forging of signatures without knowledge of the appropriate private key, and so the Byzantine node X cannot carry it out.

3.4.3 Cryptography and Trusted Setups Matter!

The fact that there are things you can do with the PKI assumption (the main result of last lecture) that you provably cannot do without it (the main result of this lecture) is super-interesting. We now understand that assumptions about the existence of secure digital signature schemes and public key distribution fundamentally affects what you can and cannot do with a consensus protocol.

For a point of contrast, think about algorithms. If you're trying to design a faster algorithm for the minimum spanning tree problem, who cares what kind of cryptography exists? Or suppose you in the middle of tackling an *NP*-hard problem like the traveling salesman problem, and I hand you on a silver platter a secure digital signature scheme (or some more fancy cryptographic primitive). You would stare at me quizzically: "What am I supposed to do this this?" Whereas, for the design of distributed protocols, cryptographic assumptions fundamentally change the game and enable solutions that otherwise would not exist.

Way back in Section 3.1 we mentioned that one purpose (among many) of impossibility results is to indicate when you're working in too demanding a model, with too few assumptions. This lecture's main result is a textbook example—it shows that if you want a consensus protocol that is robust to an arbitrary number of faulty nodes,

you have no choice but to get your hands on a secure digital signature scheme and figure out how to get everyone's public keys to everyone else (or pull off some other trusted setup assumption that makes simulation strategies infeasible for Byzantine nodes). The impossibility result in the next two lectures (in which we drop the assumption of the synchronous setting) is another textbook example—though rather than educating us about the need for trusted setup assumptions, it will guide us toward necessary assumptions on the reliability of the underlying communication network.

Bibliography

[1] M. J. Fischer, N. A. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986. URL: https://groups.csail.mit.edu/tds/papers/Lynch/FischerLynchMerritt-dc.pdf.

[2] R. L. Graham and A. C. Yao. On the improbability of reaching Byzantine agreements. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC)*, pages 467–478, 1989. URL: https://mathweb.ucsd.edu/ ronspubs/89_08_byzantine.pdf.

[3] MM. 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.

[4] T. Roughgarden. *Algorithms Illuminated, Part 4: Algorithms for NP-Hard Problems*. Soundlikeyourself Publishing, 2020.

[5] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1936. Erratum: 43:544–546, 1937. URL: https://www.astro.puc.cl/ rparra/tools/PAPERS/turing_1936.pdf.

Chapter 4

The Asynchronous Model and the FLP Impossibility Theorem (1)

4.1 Relaxing the Synchronous Assumption

The chapters 4 and 5 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 chapter series. Understanding it will be a feather in your cap and put you in quite rarified company.

4.1.1 Recap and Context

Chapter 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 DolevStrong 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 chapters (waiting until chapter 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 Chapter 3, the PKI assumption fundamentally changes whether or not the Byzantine broadcast problem is solvable in the synchronous model (with $f \geqslant \frac{n}{3}$, where n is the number of nodes and f is the maximum number of Byzantine nodes). In the asynchronous model studied in these lessons, it turns out not to matter (the FLP impossibility theorem holds even with the PKI assumption). The third assumption is that at least some of the nodes run the protocol honestly. By honest that means without any intentional or unintentional deviation from what the protocol prescribes and we'll use a parameter f to denote an upper bound on how many nodes cannot be honest and how many nodes might deviate from what the protocol prescribes. In a blockchain setting, the most appropriate model for a node which is not honest is to assume it's controlled by an attacker so we call it something known as a byzantine node. We basically make no assumptions about the behavior of byzantine nodes we might assume something like they can't break cryptography but they can try to manipulate the protocol in any way that they like. The fourth assumption (really, two subassumptions) is the one we're keen to relax in this chapter: (i) all nodes share a global clock (common notion of time); (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 firsthand 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.

4.1.2 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 DolevStrong 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, and so on. The behavior and guarantees of the modified protocol then match those of the original protocol in the original model.

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 chapter 6. But first, let's explore what's possible with only the most minimal assumptions about the reliability of message delivery.

4.2 The Asynchronous Model

Informal description. The asynchronous model of communication represents the polar opposite of the synchronous model, with the two subassumptions replaced by nonassumptions. First, there will be no shared global clock, and thus no (even approximately) shared notion of time. 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:

Note:-

The Asynchronous Model

- M denotes the pool of outstanding (not-yet-delivered) messages (initialized to $\{(r, \perp)\}_{r=1}^{n}$) The model is completely event driven and there's no notion of time. Messages are only going to be sent in response of messages that are received.
- while(TRUE):
  - an arbitrary message $(r, m) \in M$ is delivered (to recipient r, m is the message content); (subject to the eventual delivery constraint)
  - r can add any number of messages to M. In this model, we make no assumptions about message delays and in particular there's no promises about the ordering for example if one message was sent prior to another, there's no guarantee that it's going to be received prior to another

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.

Another assumption that we make is that all messages are eventually delivered; it could be an arbitrarily large but finite amount of time. This is sort of a minimal assumption for the model to be interesting otherwise for instance a bunch of honest nodes could just be starved till the end of time and in this case, obviously we cannot maintain consistency and liveness.

A conspiracy of adversaries. In Chapters 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 Chapter 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.

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? So, if you're designing a consensus protocol, this would be the dream because we like to have a consensus protocol that has nice qualities even under these completely minimal assumptions. 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.

4.3 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 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 Chapters 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 chapter'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:

Note:-

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.

4.4 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 4.4.1 FLP Impossibility Theorem [2]

For every $n \geqslant 2$, 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 of the properties that we want all of the time. 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).
- 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!
- The statement of Theorem 4.4.1 above actually undersells the FLP impossibility theorem, which holds even with a single crash fault. (As discussed in Chapter 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. To think about it, there's no solution to Byzantine problem without a solution for the Byzantine broadcast either because from the latter we can build the former.
- 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 4.4.1 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.
- 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 Chapter 6)

The question is why is the FLP impossibility theorem true?

4.5 Proof of Theorem 4.4.1: Configurations

Like the PSL-FLM impossibility result in chapter 3 (for Byzantine broadcast in the synchronous model with $f \geqslant \frac{n}{3}$ 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 4.1). 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

(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., $V = \{0, 1\}$).

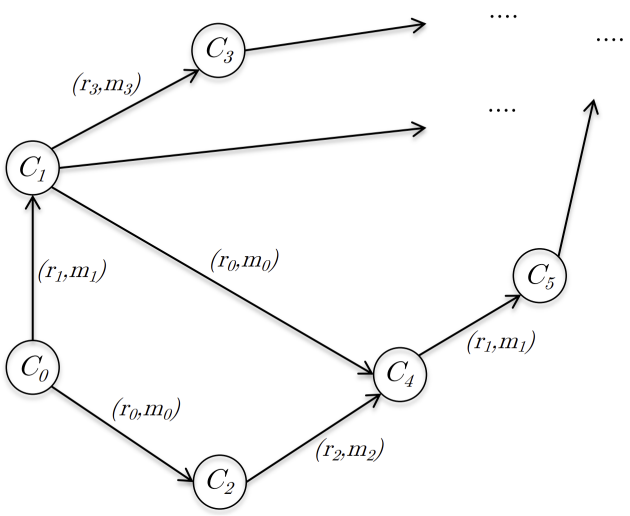


Figure 4.1: A run of a protocol can be visualized as a walk through a directed graph, with vertices corresponding to configurations and edges to message deliveries.

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 4.5.1: 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 4.1, 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. Lemma 4.6.1 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 5.1.1 will then show how to exhibit a new ambiguous configuration C_{i+1} from an old one C_i. Applying Lemma 4.6.1 once and then Lemma 5.1.1 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 4.6.1 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.

4.6 Proof of Theorem 4.4.1: Initial Ambiguity

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

Lenma 4.6.1 An Initial Ambiguous Configuration

For every deterministic protocol π that satisfies agreement and validity upon termination (with $f = 1$ and some $n \geqslant 2$), 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 $n1$ of the 2^n initial configurations to identify an ambiguous one.

Proof: 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 $X_0, X_1, X_2, \cdots, X_n$(Figure 4.2). (In X_i , the first i nodes have a private input of 1 and the last ni 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 of Definition 4.5.1, 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.

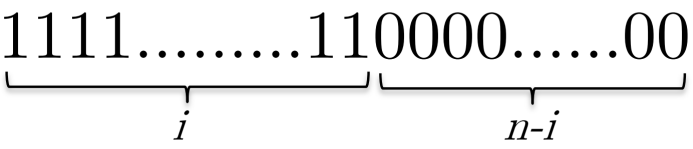


Figure 4.2: In the initial configuration X_i, nodes $1, 2, \cdots, i$ have a private input of 1 and nodes $i+1, i+2, \cdots, n$ have a private input of 0.

We claim that one of the intermediate configurations $X_1, X_2, \cdots, X_{n1}$ must be an ambiguous configuration (in which case, we'll be done with the proof). To see this, consider the smallest value of $i \geqslant 1$ 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_{i1} 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:

1. any strategy that forces the all-one outcome (which must exist, given that X_i is not a 0-configuration);
2. the strategy in which it pretends its private input is actually 0 and otherwise follows the protocol π honestly.

With strategy (2), the trajectory of the protocol π is exactly the same as its trajectory from the initial configuration X_{i1} 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_{i1} 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.

Bibliography

[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.

Chapter 5

The Asynchronous Model and the FLP Impossibility Theorem (2)

5.1 Proof of Theorem 4.4.1: Reduction to Lemmas 4.6.1 and 5.1.1

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, 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 4.6.1 gets the sequence started, and applying Lemma 5.1.1 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 5.1):

> **Lenma 5.1.1** Extending an Ambiguous Sequence
>
> Fix a deterministic protocol π that satisfies agreement and validity upon termination (with $f = 1$ and some $n \geqslant 2$). 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:
>
> 1. the last step of the sequence is the delivery of (r, m);
> 2. the end of the sequence is an ambiguous configuration C_{i+1}

Another way of saying this is Lemma 5.1.1 guarantees that there is a way to eventually deliver the message in question (r, m) while remaining in an ambiguous configuration.

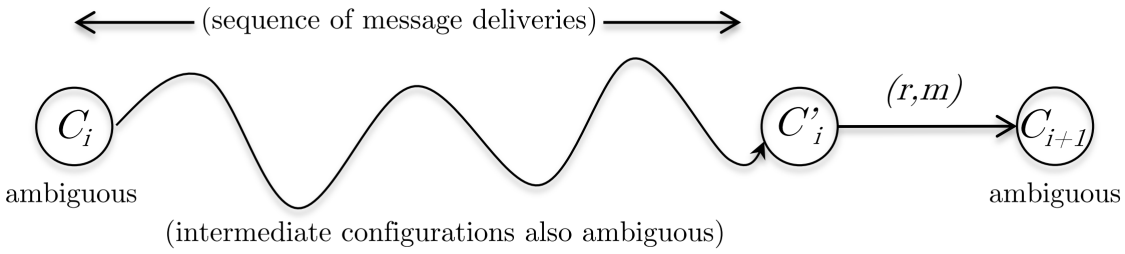


Figure 5.1: Lemma 5.1.1: extending a sequence of ambiguous configurations (from C_i to C_{i+1}) while delivering the message (r, m) last.

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 4.6.1 and then deliver an infinite sequence of dummy messages (messages of the form $(r, \perp)$, see Section 4.2). 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 4.2, we insisted on the 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 5.1.1 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 4.6.1 and (the as-yet-unproven) Lemma 5.1.1 in tandem imply the FLP impossibility theorem:
Proof of Theorem 4.4.1: Fix a deterministic protocol π that satisfies agreement and validity upon termination (with $f = 1$ and some $n \geqslant 2$). No prizes for guessing the first step: invoke Lemma 4.6.1 to choose an initial configuration C_0 that is ambiguous (which must exist).
Presumably we then want to invoke Lemma 5.1.1 over and over. Each invocation asks us to pick a message (r, m) in the current message pool—how should we choose? Putting Lemma 5.1.1 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 5.1.1 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 4.6.1;
- for $i = 0, 1, 2, \cdots$:
  - 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 5.1.1 (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 5.2)—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 4.6.1 and 5.1.1, all of the C_i's—and hence, also all of the intermediate configurations—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 to be delivered by configuration $C_{i+|M|}$ at the latest. (Each invocation of Lemma 5.1.1 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.)

Now we know that Lemma 4.6.1 is true and that Lemmas 4.6.1 and 5.1.1 imply the FLP impossibility result. The final order of business is to prove Lemma 5.1.1.

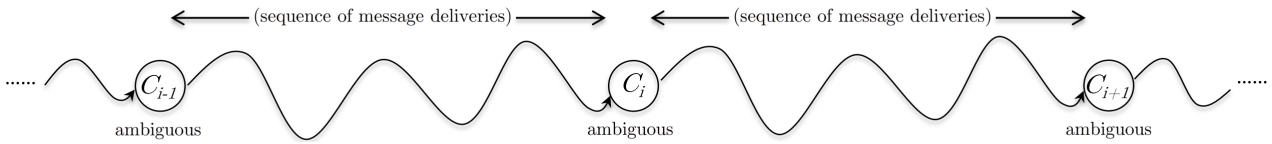


Figure 5.2: Repeated applications of Lemma 5.1.1 produces an arbitrarily long sequence (equivalently, sequence of sequences) of ambiguous configurations.

5.2 Proof of Theorem 4.4.1: Extending an Ambiguous Sequence

To prove Lemma 5.1.1, fix a deterministic protocol π that satisfies agreement and validity upon termination (with $f = 1$ and some $n \geqslant 2$), 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).

5.2.1 Proof Setup

Next we classify configurations into three categories. Like the classification in Section 4.5, this one will be defined with respect to the fixed protocol π. Unlike the one in Section 4.5, 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 5.2.1: 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;
- 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^*-, and 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). If you think about it, ambiguous* is really a predecessor of an ambiguous configuration by definition and remember predecessors of ambiguous configurations must themselves be ambiguous because again ambiguity is never introduced, it's only ever resolved. So in other words, an ambiguous star configuration is the special case of an ambiguous configuration that remains ambiguous after the delivery of the message (r, m).
With this new terminology, we can crisply rephrase our goal and one of our standing assumptions:

- conclusion of Lemma 5.1.1: 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.)

5.2.2 Hunting for a Non-0^*-Configuration

Hunting via breadth-first search. The proof plan for Theorem 4.4.1—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 4.1). 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 5.2.1, 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 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 $X \longmapsto Y$ transition); see Figure 5.3. 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 5.4). 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.

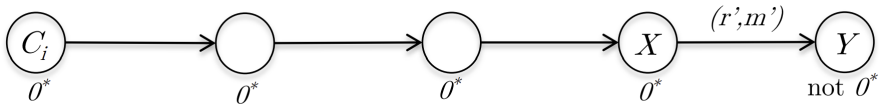


Figure 5.3: Let Y denote the non-0^*-configuration that can be reached from C_i via the delivery of the fewest number of messages (none of which are (r, m)), and X its predecessor configuration on this shortest path.

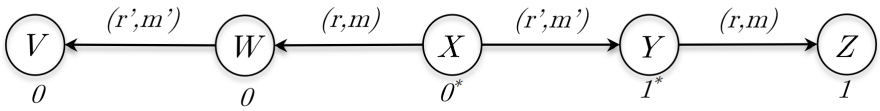


Figure 5.4: If Y is a 1^*-configuration, the order of the delivery of the messages (r, m) and (r', m') starting from configuration X dictates whether the protocol halts with the all-zero or the all-one outcome.

A pivotal pair of messages. The key takeaway from Figure 5.4 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 4.6.1.

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 4.6.1 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.

5.3 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 chapter, let's not forget the forest for the trees.

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) 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 Chapter 7) favoring consistency when under attack and longest-chain protocols (introduced in Chapter 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, Chapter 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 Chapter 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 chapter, Chapter 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 chapters are among the greatest hits of computer science. And even though our eyes are focused primarily on the future in this chapter 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.

Chapter 6

The Partially Synchronous Model, 33%, and the CAP Principle

6.1 Overview

This chapter has three goals. The first is to introduce you to the "partially synchronous" model of computation, which interpolates between the synchronous model (studied in chapters 2 and 3) and the asynchronous model (the subject of chapters 4 and 5). If you remember only one model of message delivery, this should probably be the one. It's a "sweet spot" model in that its assumptions are weak enough to make it relevant to the design of Internet-scale distributed protocols, but also strong enough so that interesting protocols with provable guarantees exist.

The second goal is to establish limitations on what we can hope for in the partially synchronous model. In particular, we'll prove an impossibility result that rules out strong positive results in the case that at least one-third of the nodes can be Byzantine (i.e., $f \geqslant \frac{n}{3}$, where n is the number of nodes and f is an upper bound on the number of Byzantine nodes). Whenever you hear about a critical "33%" threshold in a blockchain whitepaper or technical discussion, it boils down to this impossibility result. As a bonus, the proof is not too bad (certainly easier than the proof of the FLP impossibility result) and the basic intuition behind is something that you can retain for a long time to come.

The third goal of this chapter is to discuss a famous principle from distributed systems, the "CAP Principle." The letters stand for "consistency," "availability," and "partition tolerance," and the principle states that you must pick two of these three properties (you can't have them all). The CAP principle superficially resembles the FLP impossibility theorem for the asynchronous model, and it's important to understand the differences between them. After this chapter, when you hear someone appeal to the CAP principle to justify their system's weaknesses, you'll be in a position to assess whether they know what they're talking about.

6.2 The Story So Far

To set the stage for the partially synchronous model, let's review the two models of communication that we've studied thus far (in chapters 2 and 3 and chapters 4 and 5, respectively):

Note:-

Recap: Synchronous vs. Asynchronous Model

Synchronous model:

- shared global clock;
- a priori known bound Δ on the maximum message delay;
- good news: strong positive results possible (e.g., from chapter 2, with the PKI assumption, the Dolev-Strong protocol, and the consequent SMR protocol that can tolerate 99% Byzantine nodes);

- bad news: overly strong assumption (rules out network outages and attacks of unexpected durations)

Asynchronous model:

- no global clock;
- no assumptions on message delays other than that every message eventually gets delivered;
- good news: with such weak assumptions, any positive result (i.e., consensus protocol with provable guarantees) is automatically interesting and impressive;
- bad news: by the FLP impossibility theorem, no deterministic protocol can solve Byzantine agreement (or state machine replication) with the threat of even a single crash fault.

Synchronous model. In the synchronous model, there's a shared global clock: all nodes agree at all times on the current time step (with no communication needed). There's also an assumed bound Δ on the maximum number of time steps that a message might get delayed (any message sent in a time step t is guaranteed to arrive by time step $t + \Delta$, if not earlier). The value of the parameter Δ is known upfront, meaning that the description of a protocol can depend on it.
We saw in Chapter 2 that strong positive results are possible in the synchronous model (at least under the PKI assumption), with the Dolev-Strong protocol for Byzantine broadcast resilient to even 99% of the nodes being Byzantine. Using our standard rotating leaders trick, leads to an equally fault-tolerant protocol for the (multi-shot) state machine replication (SMR) problem.
On the other hand, while the shared global clock assumption is perhaps somewhat palatable, the known bound on worst-case message delay is an unreasonable assumption for protocols that operate at the scale of the Internet. The Internet does suffer long outages on occasion, and blockchain protocols that secure billions of dollars of value will inevitably have to deal with long-lasting attacks.

The asynchronous model. The asynchronous model is the polar opposite of the synchronous model, with no global shared clock and no guarantees whatsoever about message delivery (other than the minimal assumption that every message eventually gets delivered, after some finite delay). Because this model's assumptions are so weak (requiring only a barely functioning communication network), any protocol with strong provable guarantees would automatically be interesting. Unfortunately, the FLP impossibility theorem showed us that this model throws out the baby with the bathwater, in the sense that no good consensus protocols exist (without resorting to randomization or extra timing assumptions), even under the threat of a single Byzantine (or even crash fault) node.
The synchronous and asynchronous models are probably the two most natural models of communication to write down. But neither model guided us toward what we want: a consensus protocol that might plausibly be useful at the Internet scale. It's thus clear that we need a third model that interpolates between the synchronous and asynchronous models, with assumptions weak enough to be relevant to Internet-scale protocols but strong enough such that useful consensus protocols with provable guarantees exist. Such a model is the subject of this chapter.

6.3 The Partially Synchronous Model

6.3.1 Intuition: Recovery After an Attack/Outage Eventually Ends

We rejected the synchronous model on the grounds that network outages and attacks of unexpected durations will at some point occur. But outages and attacks must end eventually, right? Maybe it takes a few hours, or maybe a few days, but surely we're interested in a world where at some point the world becomes "normal" again. A key idea in the partially synchronous model is to explicitly declare some periods of time "normal" (and thus well-modeled by the synchronous model) and other periods "under attack" (and thus appropriately modeled by the asynchronous model). The goals then would be:

- (sanity check) achieve everything you want (e.g., safety and liveness) during "normal operation conditions";
- (stress test) don't break too badly when under attack (e.g., give up only safety, or only liveness); We will be identifying an attack phase with the asynchronous model that we discussed in the last two chapters. The FLP impossibility result says that if we're in an asynchronous phase and we're under attack during that

time we cannot have everything that we want, we cannot have both safety and liveness but we'd still like to aspire to giving up as little as possible while we're under attack so at least we keep safety or at least we keep aliveness.

- (recovery) recover quickly after an attack once the network returns to normal operating conditions. After an attack ends and after we switch from asynchrony to a synchronous setting, we would like to then have the guarantees we expect in the synchronous setting for example safety and liveness.

One could imagine a model in which there's an arbitrary number of alternative synchronous and asynchronous phases. We'll work with the most minimal version of this idea in which there's a single synchronous phase and a single asynchronous phase. (Chaining together several copies of the minimal model essentially recover the case of an arbitrary number of alternations.) To study recovery after an attack (the third goal above), it makes sense to have the asynchronous phase first, followed by the synchronous phase.

6.3.2 The Formal Definition

Next is (one of) the formal definition of the partially synchronous model. This definition is due to Dwork, Lynch, and Stockmayer; the same authors proved some fundamental possibilities and impossibility results in the same paper.

Timing assumptions. The first assumption is that, like in the synchronous model, there is a shared global clock. That is, whether under attack or in normal operating conditions, all nodes automatically agree (without any communication) on what the current time step is. While we'd rather not have this assumption, it's definitely the more palatable of the two assumptions that we made in the synchronous model (and you can start imagining ways of approximating it in practice).

The synchronous phase. Second, there will be a parameter Δ that specifies the maximum delay (in time steps) that a message might suffer in the synchronous phase. During the initial asynchronous phase, the parameter Δ plays no role. Like in the synchronous model, in (this version of) the partially synchronous model, we assume that the value of Δ is known up front and that the protocol description can depend on it. For example, you could think of each time step as representing one millisecond and Δ equal to 1000 or 2000 (1 or 2 seconds).

GST: the transition point. Finally, there is a parameter that dictates when the underlying communication network switches from synchronous to asynchronous mode, called the global stabilization time or the GST. Unlike the parameter Δ, the parameter GST is not known upfront and so a protocol description cannot depend on it. In other words, a protocol is responsible for providing its guarantees no matter what the GST might be. A protocol is not informed when the GST occurs, so it effectively must detect its passing automatically (and resume normal operation accordingly). Therefore, it's very important that the global stabilization time is unknown to the protocol; the protocol should work simultaneously for all possible GSTs. Taken together, the parameters GST and Δ impose the following constraints on message delivery (for the a priori known parameter Δ and whatever value of the GST the message delivery adversary happens to choose):

1. (messages sent in asynchronous phase) if a message is sent at time step t with $t \leqslant GST$, then it is received by the intended recipient at or before time step $GST + \Delta$ (i.e., as if it was sent at time GST);

2. (messages sent in synchronous phase) if a message is sent at time step t with $t \geqslant GST$, then it is received by the intended recipient at or before time step $t + \Delta$.

Because the GST must be finite, every message that is sent must eventually be delivered. In this sense, the partially synchronous model is a special case of the asynchronous model. The asynchronous model is strictly more general, as message delivery there does not need to obey any quantitative constraints such as the ones above. The value of the GST must be unknown to the protocol. In effect, the adversary controls message delivery and chooses it as a function of the chosen protocol. One reason is conceptual: the whole point of the partially synchronous model is to force us to reckon with network outages and attacks of unexpectedly long duration. Another reason is technical: were the GST known upfront, all possibility results for the synchronous model (e.g., the Dolev-Protocol) would port over immediately with a stalling pre-processing step (the protocol could simply wait until the GST and then proceed as in the synchronous model).

Protocols. We'll later prove an impossibility result for the partially synchronous model, and for that, we'll need a precise definition of what a protocol can and cannot do. In the asynchronous model, a protocol defined the action of a node (i.e., which messages to send) as a function of what the node knows, namely its private input and the sequence of messages it has received thus far. Because the partially synchronous model introduces some timing assumptions, nodes have additional information: the current time step and the precise time step at which each received message arrived. A protocol is then defined as a function from the node's current knowledge (private input, current time step, messages received, and their arrival times) to an action (which messages to send out and to whom). Unlike in the asynchronous model, protocols in the partially synchronous model can implement timeouts (e.g., ignoring messages that arrive many time steps after they were expected). Next chapter we'll see how the Tendermint protocol takes advantage of this additional power.

The "unknown Δ" model. Somewhat confusingly, there are two conceptually distinct versions of the partially synchronous model (both defined by Dwork, Lynch, and Stockmeyer). Throughout this chapter series, we'll focus on the "GST" version of the partially synchronous model that we've been discussing thus far. Elsewhere, you might also encounter a second version of the model, so let's discuss that briefly next.

The second version of the model connects strongly to our discussion at the beginning of chapter 4. There, we made a naive attempt at relaxing the synchronous model (with $\Delta = 1$) to a model with variable message delays (anywhere from 1 up to a known upper bound Δ). We then noticed that any possibility result for the $\Delta = 1$ model extends automatically to the general (but known) Δ case through "time inflation," with all nodes always waiting Δ time steps in between consecutive actions to make sure all messages sent by honest nodes are received in time. Dissatisfied, we pined for a consensus protocol whose speed is not dictated by the worst-possible message delay, and rather adapts automatically to the network speed and is fast whenever the underlying communication network is fast. This is exactly the idea behind the second version of the partially synchronous model.

In the "unknown Δ" version of the model, there's no GST, and no transition between an asynchronous and a synchronous phase. Messages will literally always get delivered within Δ time steps, just like in the synchronous model. Unlike the synchronous model (and the "GST version" of the partially synchronous model, however, the parameter Δ is not known in advance and the protocol description cannot depend on it. In other words, a protocol must work simultaneously in every instantiation of the synchronous model (i.e., whatever the network speed/parameter Δ).

Why two versions? The two versions of the partially synchronous model both seem natural—the "GST version" encodes the goal of quick recovery after an outage or attack, and the "unknown Δ" version the goal of operating at network speed (whatever that may be)—but also rather different. Why do they go under the same name? One reason is that the original paper introduced both versions under the umbrella of "partial synchrony." This tradition has persisted to the present day largely because possibility and impossibility results that hold in one of the two versions tend to hold also in the other version. The two versions are thus roughly (but not formally) equivalent for our purposes. It's a good exercise to rework the proof of Theorem 6.5.1 (stated for the "GST version") so that the same impossibility result holds for the "unknown Δ" version of the partially synchronous model. Similarly, you might want to think about how to rework the Tendermint protocol (covered in the forthcoming chapter 7) so that its consistency and liveness guarantees (with $f < \frac{n}{3}$) hold also in the "unknown Δ" model. We won't discuss this version of the model again in these lessons.

6.4 Goals for a Consensus Protocol

In the partially synchronous model, what should we ask for from a consensus protocol? For example, when should we deem a protocol a "solution" to the Byzantine agreement problem in the partially synchronous model? As a quick reminder (from chapter 4), in the Byzantine agreement problem, every node i has a private input v_i (drawn from some set V) and the goals are:

Note:-

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.

The FLP impossibility result (chapters 4 and 5) tells us that no consensus protocol guarantees all of these properties in the asynchronous model. The partially synchronous model is less general that the asynchornous model (on account of the constraints on message delivery spelled out in Section 6.3.2), so the FLP impossibility result does not immediately apply. It does apply during the asynchronous phase of the partially synchronous model, and so every protocol that guarantees validity and agreement on termination must in some cases not terminate until after the global stabilization time. In other words, if no violations of validity or agreement are allowed, a protocol cannot guarantee liveness during the asynchronous phase. By the same reasoning, for the SMR problem, no protocol that always guarantees consistency can guarantee liveness prior to the global stabilization time. Post-GST, however, there is hope of guaranteeing both safety and liveness.

Summarizing, the best we could hope for would seem to be:

- at least one of the safety or liveness during the asynchronous phase;
- safety and liveness during the synchronous phase (beginning not long after the GST).

What should we give up during the asynchronous phase? Arguably the most natural choice, and the only choice explored in 20th-century research on the partially synchronous model, is to give up liveness and preserve safety. Safety properties state that "something bad never happens" and it's sensible to interpret "never" as "including in the asynchronous phase." Liveness properties assert "something good eventually happens," and "eventually" might naturally be interpreted as "after the GST."

Note:-

Traditional Goals in the Partially Synchronous Model

1. Safety: Safety holds always, even in the asynchronous phase.
2. Eventual liveness: Not long after the GST, safety and liveness both hold (e.g., consistency and liveness for SMR);

Many blockchain consensus protocols adopt these same two goals, and this will be our focus in this and the next chapter. That said, one interesting aspect of longest-chain consensus (discussed in chapter 8) is that it makes non-traditional compromises in periods of asynchrony, favoring liveness over safety.

6.5 The Big Result

When are safety and eventual liveness achievable in the partially synchronous model for the Byzantine agreement problem? Turns out there's a remarkably crisp answer to this question, with the magical threshold being 33%. (Remember that n always denotes the number of nodes and f the upper bound on the maximum number of Byzantine nodes.)

Theorem 6.5.1

There exists a deterministic protocol for the Byzantine agreement problem that satisfies agreement, validity, and eventual (post-GST) liveness in the partially synchronous model if and only if $f < \frac{n}{3}$.

This is two results in one, a possibility result and a matching impossibility result. We'll take up the impossibility result—that with $f \geqslant \frac{n}{3}$, no Byzantine agreement protocol can achieve safety and eventual liveness. In chapter 7 we'll study the Tendermint protocol, which does in fact achieve safety and eventual liveness provided $f < \frac{n}{3}$. Some comments:

1. Tendermint is only one protocol among many that achieves optimal fault tolerance in the partially synchronous model. (For example, the original paper also describes a solution.) Tendermint is an obvious one for us to single out, as it is used in several major blockchain protocols.
2. Theorem 6.5.1 holds whether or not we make a PKI assumption. The impossibility result we'll prove in this chapter holds even under the PKI assumption (Byzantine nodes won't need to forge any signatures to carry out their devious strategy). The Tendermint protocol makes use of a PKI assumption, but the same positive result is possible (with a different protocol) without it.

3. Like the FLP impossibility theorem from chapters 4 and 5, this impossibility result is stated for the Byzantine agreement problem but applies equally well to the SMR problem.6 The Tendermint protocol in chapter 7 solves the SMR problem, which can in turn be used to construct (via the reduction in footnote 6) an equally fault-tolerant Byzantine agreement protocol.

4. You frequently see distributing computing experts write the f ¡ n/3 assumption as, equivalently, $n \geqslant 3f + 1$. This is a famous, almost meme-like inequality in distributed computing—if no conference has yet printed T-shirts with "$n \geqslant 3f + 1$" on the front, it's long overdue!

5. Whenever you hear a blockchain person or whitepaper refers to a "33% threshold"— generally to justify why a protocol isn't more robust to attacks than it already is, it boils down to the one in Theorem 6.5.1.

6. If you remember only one result from this boot camp on permissioned consensus (chapters 2–7), Theorem 6.5.1 would be a good one. It is the culmination of our study of the 20th-century literature on the design and analysis of consensus protocols.

6.6 Intuition for Impossibility

The goal of this section is to provide you with reasonably accurate intuition about why the "only if" direction of Theorem 6.5.1, the impossibility result, is true. This intuitive argument should demystify where the magical "33% comes" from, which in hindsight should seem preordained. The informal argument is compact enough that you should be able to remember it for a significant period of time. Because the result is so important, I must also offer you full proof of this, see Section 6.7. The role of unbounded message delays. First, let's talk about the proof strategy and the ingredients we expect to see in the argument. Theorem 6.5.1 is not the first "33%" impossibility result that we've seen. The PSL-FLM result from Chapter 3 established the same threshold for the Byzantine broadcast problem in the synchronous model when there is no PKI assumption. Are these the same "33%"? Perhaps Theorem 6.5.1 can somehow be reduced to the PSL-FLM result that we worked so hard to prove in Chapter 3? Maybe there's no real difference as to the degree of fault tolerance possible in the synchronous and partially synchronous models?

In fact, completely different forces are driving the two impossibility results. Remember that the PSL-FLM impossibility result does not hold with the PKI assumption, the Dolev-Strong protocol (chapter 2) solves the Byzantine broadcast problem (i.e., satisfies agreement, validity, and termination) in the synchronous model even when 99% of the nodes are Byzantine. As a consequence, its proof must crucially hinge on the lack of PKI. This dependence showed up in a subtle way, the proof uses clever strategies for Byzantine nodes that involve the simulation of four honest nodes (in the "hexagon thought experiment"), and these strategies are infeasible under the PKI assumption (because it would require forging signatures without knowledge of the appropriate private key, which with PKI we assume is impossible). As mentioned in the previous section, Theorem 6.5.1 is true whether or not we make the PKI assumption. Presumably, then, the proof of Theorem 6.5.1 in Section 8 will not by making use of strategies for Byzantine nodes that involve the simulation of honest nodes. This was the only trick used in the proof of the PSL-FLM impossibility result, so something else must be driving the "33%" in Theorem 6.5.1. Given that the result is for the partially synchronous (as opposed to synchronous) model, presumably the proof will be driven by the threat of potentially unbounded message delays.

Intuition, part 1: plausibly deniable silent treatment. First, even in the synchronous model, in order to satisfy termination, an honest node must be prepared to halt with an output even if it hasn't heard anything at all from some of the other nodes. After all, one strategy for the Byznatine nodes is to give the honest nodes the silent treatment and never send out any messages. Because up to f nodes may be Byzantine, an honest node can only count on hearing from nf nodes (counting itself) before needing to make a decision.

Intuition, part 2: wolves in sheep's clothing. Because we're now in the partially synchronous model, there's an equally plausible explanation for why an honest node might not have ever heard from a set of other nodes, maybe those nodes are actually honest and sent all the messages that they were supposed to send, but all of their messages have been delayed for a very long time. (This is only possible before the global stabilization time, but remember that the GST is unknown to the protocol can be arbitrarily large.) This cannot happen in the synchronous model. If you don't hear the messages that you expect coming in from some node, the only possible explanation is that that node is Byzantine. Thus, the real bummer is that when an honest nodes takes action after hearing from only nf nodes (as it must), for all it knows f of the voices its hearing belong to Byzantine

nodes (with all the as yet unheard from nodes honest but suffering massively delayed messages). Like wolves in sheep's clothing, these Byzantine node might lead the honest node astray by feeding it bad information (e.g., pretending their private input was 1 rather than 0). Intuitively, if at least 50% of these nf nodes are Byzantine, there's no way for the honest node to determine whom to believe. (If a strict majority of them are honest, one might hope that some kind of majority vote could indicate the appropriate action.) This leads to the requirement that $f < \frac{1}{2}(nf)$ or, equivalently, that $f < \frac{n}{3}$.

Note:-

Demystifying the 33%

1. An honest node can only wait to hear from nf nodes (counting itself) before taking action. By the termination requirement and the fact that Byzantine nodes may never respond.
2. To avoid getting tricked, a strict majority of these nf nodes must be honest (and so $f < \frac{1}{2}(nf)$ or $f < \frac{n}{3}$). The f as-yet-unheard-from nodes may be honest (with their messages massively delayed), meaning f of the nf nodes that have been heard from may yet be Byzantine.

This informal argument highlights the challenge of de facto collusion between the Byzantine nodes and adversarial message delivery. Both Byzantine nodes and the adversary controlling message delivery have to the power to enforce silence from f nodes for an arbitrarily long period of time, and an honest node cannot distinguish whom is the culprit. In effect, each Byzantine node winds up knocking out two honest nodes, one honest node whose contributions are ignored (because they are massively delayed and it gets mistaken for a Byzantine node) and a second honest node whose received messages are cancelled by conflicting messages sent by a Byzantine node. Intuitively, we need one honest node still standing after all these knockouts, meaning $(nf)2f > 0$ or, equivalently, $f < n/3$. All this is accurate and hopefully retainable intuition about why (the "only if" direction of) Theorem 6.5.1 is true, but none of it constitutes an actual convincing proof. So let's get to it!

6.7 Proof of Theorem 6.5.1 ("Only If" Direction)

Statement of impossibility result: Recall what we're trying to prove: in the partially synchronous model (defined in Section 6.3.2), with or without the PKI assumption, for every f and n with $f \geqslant \frac{n}{3}$, no Byzantine agreement protocol achieves the goals spelled out in Section 6.4: agreement (always), validity (always), and termination (eventually, possibly only after the GST).

A simple case capturing all the complecxity. We're going to focus on the simplest possible case of the impossibility result, with $n = 3$ and $f = 1$ (three nodes, one of which might be Byzantine). (We did the same thing back in chapter 3 for the PSL-FLM impossibility result for Byzantine broadcast in the synchronous model without the PKI assumption.) Your first reaction might be that this assumption trivializes the result, but the truth is the exact opposite—this special case already captures all of the complexity and nuance of the general impossibility result. A good exercise is to rework the following argument so that it applies to your favorite choice of n and $f \geqslant \frac{n}{3}$. Call the three nodes "Alice (A)," "Bob (B)," and "Carol (C)" and see Figure 6.1. Toward a contradiction. Suppose there is a Byzantine agreement protocol π that satisfies agreement (always), validity (always), and termination (eventually). Consider the scenario in which Alice is honest and has a private input of 1, Carol is honest and has a private input of 0, and Bob is Byzantine (and hence his private input doesn't matter).

The adversaries' collusive strategy. Imagine that Bob engages in the canonical Byzantine ploy of sending inconsistent messages to different honest nodes, while aided and abetted by the adversary controlling message delivery. Specifically:

- The adversary controlling message delivery delays all messages between Alice and Carol for a long time. (How long? See below.)
- Bob interacts with Alice as if he has a private input of 1 and never received any messages from Carol.
- Bob interacts with Carol as if he has a private input of 0 and never received any messages from Alice.

Note that Bob is perfectly capable of carrying out this strategy (whether or not we're working with the PKI assumption). All he has to do is run (two copies of) π with the above fabricated private inputs and message

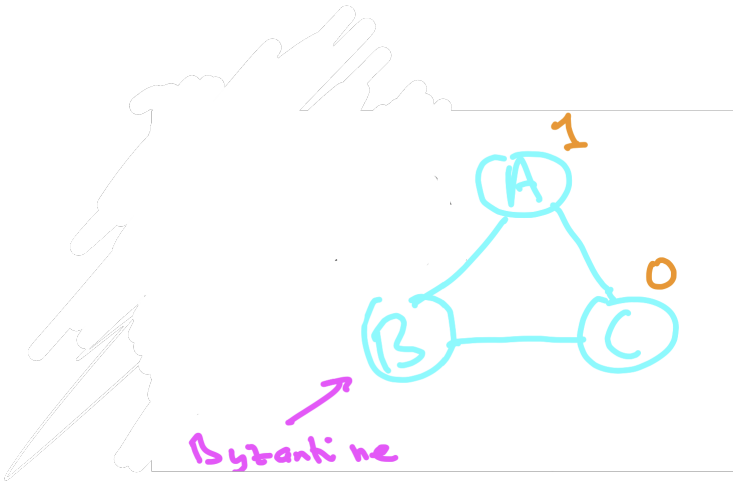


Figure 6.1: Proof of impossibility: $f \geqslant \frac{n}{3}$

sequences. Because the GST in the partially synchronous model can be arbitrarily large (as a function of the specific protocol π), the adversary controlling message delivery has the power to delay messages between Alice and Carol for as long as it wants (as they as they are eventually delivered). Two catch-22s. Now adopt Alice's perspective, being fed a constant stream of (mis)information from Bob and silence from Carol. Alice is perfectly aware of the possibility of the scenario above, that Bob is Byzantine and using the above strategy when Carol's honestly sent messages are stuck in a pre-GST limbo. Unfortunately, there's an equally plausible explanation for what Alice is seeing: perhaps Bob is honest and really does a private input of 1, while Carol is the Byzantine one and is providing both Alice and Bob with the silent treatment. To hedge against the latter scenario, because π satisfies eventual termination (by assumption), Alice must eventually (by some finite time T_1, and with no input from Carol) output her answer. By validity (since in the second scenario both honest nodes have the same private input), her output must be 1. Carol finds herself in her own catch-22 situation. She's well aware that reality might be the first scenario, with Bob the Byzantine one and Alice's honestly sent messages stuck in pre-GST limbo. But an equally plausible explanation for what she's seeing is that Bob really is honest with a private input of 0, with Alice the Byzantine one who is providing both Bob and Carol the silent treatment. Because of the threat of the latter scenario, Carol has no choice but to output her answer by some finite time T_2 (with no input from Alice). By validity, because in that scenario, she and (the other honest node) Bob both have a private input of 0, her output must be 0.

Putting it all together. To complete the description of the collusion between the Byzantine node and the adversarial message delivery, assume that all messages between Alice and Carol are delayed for $\max\{T_1, T_2\} + 1$ time steps (which is allowed provided the adversary chooses the GST to be at least this large). With Bob behaving inconsistently as above, and with all communication between Alice and Carol severed Alice outputs 1 (unable to rule out the case in which Bob is honest and Carol gives her the silent treatment), and Carol outputs 0 (unable to rule out an honest Bob and a silent Alice). But Alice and Carol are both honest nodes, so this outcome of the protocol contradicts the assumption that the protocol π satisfies agreement. We can conclude that no such protocol π exists, completing the proof of the "only if" direction of Theorem 6.5.1.

6.8 The CAP Principle

6.8.1 The Principle and Its Interpretations

The CAP Principle is a well-known observation in distributed systems. You might think that "CAP" indicates the first letters of three last names (as with PSL, FLM, or FLP!), but the letters actually stand for three informal properties that you would want a distributed system to satisfy. To interpret them, you might want to think about our running 20th century example in which a big company like IBM is replicating a database to achieve higher uptime.

1. 'C' stands for "consistency." It plays a similar role that consistency has played in our study of the SMR

problem. In our running example, consistency would mean that the answer to a database query should not depend on which replica it gets routed to. More generally, you would like the experience of a user of a distributed system to be indistinguishable from interacting with a centralized system (like a single database on a single server).

2. 'A' stands for "availability," which resembles the liveness property that we required for SMR protocols. In our running example, if a user of the database inserts a new entry, that change should eventually be reflected in future queries to the database. More generally, any command that a client issues to a distributed system should eventually be carried out.

3. 'P' stands for "partition tolerance." Here a partition means a long-lasting severance of all communication between two different groups of nodes (Figure 6.2). For instance, due to a long denial-of-service attack. Partition-tolerance informally means that you'd like consistency and availability to hold even in the presence of a network partition, and bears some resemblance to the asynchronous phase of the partially synchronous model defined in Section 6.3.2 (though with some important differences, detailed in Section 6.8.2).

Figure 6.2

The CAP Principle states that no distributed system possess all three properties. In particular, if a system operates in a environment with network partitions (e.g., because it operates over the Internet), it must give up on one on consistency or availability.

The argument. The reasoning behind the CAP Principle is pretty straightforward. Imagine a distributed system that is responsible for keeping track of (among other things) some variable x, and is suffering from a network partition as in Figure 6.2. For example, maybe x represents the number of times that the San Diego Padres have won the World Series (currently, 0). Suppose the Padres win the '22 Series and an excited fan issues a command to the system to increment x to 1, and suppose this command gets routed to a node i in the set A. Now consider an infinite stream of future queries about the value of x that get routed to node i. The node is in a catch-22 situation: if it ever answers "1," it could cause a violation of consistency (with all communication between A and B severed, nodes of B would presumably answer "0" to the same query), but stubbornly answering "0" for the rest of the time would violate availability. A decision tree for design. This argument may strike you as a bit trivial (and to be honest, it is), but it nonetheless suggests a useful decision tree to use when designing a distributed system:

- decide whether you want to worry about network partitions (e.g., because the system will operate over the Internet) or not (e.g., because it will operate over a secure and privately owned network);
- if not, demand both consistency and availability (somewhat analogous to our insistence on both consistency and liveness for SMR protocols in the synchronous setting);
- if so, take a hard look at your application and decide which of consistency or availability would be less painful to give up when there's a network partition.

On the last point, you can imagine that different choices might make sense for different applications. For example, something like a bank might give up on availability while insisting on consistency (e.g., the usual compromise

made by traditional database systems)—having its system go down for 24 hours while under attack is painful, but not as painful as the financial losses that could be caused by the exploitation of inconsistencies in account balances. For something like a search engine, you can imagine prioritizing availability over consistency (e.g., as offered by some NoSQL databases)—it's not a big deal if two users in different parts of the world see slightly different results for the same query, while downtime for a search engine should be minimized at all costs. Bringing the discussion back to blockchain protocol design, we'll see an analogous dichotomy play out over the next two chapters: some SMR protocols (like Tendermint, see chapter 7) give up on liveness when under attack (i.e., when in the asynchronous phase of the partially synchronous model) while others (longest-chain consensus, see chapter 8) prefer to give up on consistency.

6.8.2 FLP Impossibility Theorem vs. CAP Principle

The takeaways from the CAP Principle seem similar to those from the FLP impossibility result (chapters 4 and 5): during a network attack (an asynchronous phase or a network partition), you're forced to choose between safety/consistency and liveness/availability. Oddly, you almost never see the FLP impossibility result and the CAP Principle taught in the same course—the former shows up in theory of distributed computing courses, the latter in courses about distributed system design. But any blockchain expert should be aware of both as well as the differences between them:

1. The proof of the FLP impossibility result is hard. The argument behind the CAP Principle is not. (I say this without judgment. As you've seen, these are the facts on the ground.)

2. Why is the argument for the CAP Principle so much easier? Because the setup allows for messages (between the two sides of the partition) to be delayed forever (or simply dropped)—there's no requirement of eventual delivery, let alone a global stabilization time. This means the message delivery adversary is potentially much more powerful in the CAP case, and this extra power makes impossibility easy to argue. The FLP impossibility result shows that safety and liveness are impossible even if the adversary controlling message delivery is forced to eventually deliver every message— with the less powerful adversary, impossibility is much harder to prove. (If you remember the proof, you'll remember that we had to work quite hard to state and prove the second lemma, which is the one that ensured the satisfaction of this constraint while also extending the length of an ambiguous sequence of configurations.)

3. In principle, there's a different dimension in which the adversary controlling message delivery is more powerful in the FLP setting than the CAP setting: in the latter, this adversary is restricted to network partitions, while in the asynchronous model, this adversary can do whatever it wants (subject to the eventual delivery constraint). In hindsight, it seems that network partitions are a canonical attack and already capture much of the power of more general strategies for adversarial message delivery.

4. The adversary controlling message delivery in the CAP setting is so powerful that no other adversary is needed. That is, the argument remains valid even if all the noes are honest ($f = 0$)! By contrast, the FLP impossibility result no longer holds for the $f = 0$—at least one faulty node really is needed.

On the last point, you might recall from chapters 4 and 5 that the the proof of the FLP impossibility result can be tweaked so that the result holds even with a single crash fault (the most benign type, with the faulty node honest up until some point at which it gets unplugged forever). And a crash fault does resemble a special case of infinite message delays—for example, if the machine crashed immediately before it was about to send out a bunch of messages, the effect is the same as if those messages suffered infinite delays. In a sense, what the FLP impossibility result really shows is that even the threat of a single crash fault is enough to trigger the same conclusion (under attack, choose between safety and liveness) that you would get somewhat trivially with infinite message delays. Coming up next in chapter 7 is a possibility result which proves the "if" direction of Theorem 6.1: there is a consensus protocol that, provided less than a third of the nodes are Byzantine, guarantees safety and eventual (post-GST) liveness in the partially synchronous model. As a bonus, the specific protocol that we'll discuss (Tendermint) powers several major blockchain protocols.

Chapter 7

The Tendermint Protocol

7.1 The Story So Far

Main result of chapter. This chapter provides a possibility result for state machine replication (SMR) in the partially synchronous model that matches (in terms of the number of Byzantine nodes tolerated) the impossibility result that we proved in chapter 6. As a bonus, we'll use a well-known blockchain protocol—the Tendermint protocol—to prove the result.

The partially synchronous model. The synchronous model (chapters 2 and 3) was great for proving strong positive results (like the Dolev-Strong protocol for Byzantine broadcast)but made unrealistically strong assumptions about guaranteed message delivery. The asynchronous model (chapters 4 and 5) made pleasingly minimal assumptions but, as shown by the FLP impossibility result, throws out the baby with the bathwater, with no good (deterministic) consensus protocols possible. This led us to the "sweet spot" partially synchronous model, which explicitly declares some periods of time "normal" (and thus well-modeled by the synchronous model) and other periods "under attack" (and thus appropriately modeled by the asynchronous model). Like the synchronous model, the partially synchronous model assumes a global shared clock. The basic version of the model involves two parameters that constrain message delivery. The first is a parameter Δ that specifies the maximum delay(in time steps) that a message might suffer in the synchronous phase. This parameter is known up front, in the sense that the protocol's description (e.g., the length of a timeout)can depend on its value. The second is the global stabilization time (GST), which is the point at which the communication network transitions from the asynchronous setting to the synchronous setting. Precisely, a message sent at time step t is guaranteed to arrive by timestep $\max\{t, GST\} + \Delta$. The GST is not known up front and can be arbitrarily large (but finite)—it's the protocol's responsibility to provide guarantees no matter what GST might be.
Recall from chapter 6 the traditional goals for a consensus protocol in the partially synchronous model:

Note:-

Traditional Goals in the Partially Synchronous Model

1. Safety: Safety holds always, even in the asynchronous phase (i.e., preGST).
2. Eventual liveness: Not long after the GST, safety and liveness both hold.

Let's also remember the specific safety and liveness conditions used in the SMR problem (defined way back in chapter 2). First, recall that in the SMR problem, clients submit transactions to a network of nodes that are responsible for running a consensus protocol, with each node maintaining a local append-only sequence of transactions (its "history"). Safety and liveness then mean:

Goal 1: Consistency. No two nodes ever disagree on the relative order of two different transactions. (Ideally, they would stay perfectly in sync, but we allow some nodes to fall behind as long as they eventually catch up with the others.)

Goal 2: Liveness. Every transaction submitted to at least one node is eventually added to every node's local history. (Actually, in this and the next chapter we'll study a slightly weaker notion of liveness, see Section 6 for

details.)
Last chapter we stated the following (where, as usual, n denotes the number of nodesand f the number of Byzantine nodes):

> **Theorem 7.1.1**
>
> There exists a deterministic protocol for the SMR problem that satisfiesconsistency and eventual (post-GST) liveness in the partially synchronous model if and only if $f < \frac{n}{3}$.

Theorem 7.1.1 holds with or without the PKI assumption. In this chapter we'll prove the "if"direction under the PKI assumption, but the same positive result can be achieved without it.

7.2 Tendermint: High-Level Ideas

While Tendermint is a 21st-century protocol designed with blockchains in mind, the way that it works strongly resembles an iterated version of classical Byzantine agreement protocols from the 1980s and 1990s. This section introduces three of the main high-level ideas in the protocol, and the protocol is then detailed in Section 7.3.The three ideas outlined in this section will probably seem reasonably natural to you,but keep in mind that there are many different ways in which you could turn those ideas into a precisely defined protocol. And the devil is in the details: most approaches would produce a buggy protocol that fails consistency, eventual liveness, or both.In some areas of computer science, intuition tends to be a good guide. The design of distributed protocols is not one of those cases. You should be automatically suspicious of any proposal that is not precisely described (as we do for Tendermint in Section 7.3) or that does not come equipped with rigorous proofs of its alleged safety and liveness guarantees (as in Sections 7.4 and 7.5 for the Tendermint protocol).

7.2.1 Idea 1: Iterated Single-Shot Consensus

The first idea is to reduce the multi-shot SMR problem that we care about to single-shot consensus. Tendermint will basically iterate a Byzantine agreement-type protocol over andover again, with one invocation for each block of transactions (i.e., a batch of transactions, sequenced in some way). That is, nodes will agree on block 1 (via single-shot consensus),then on block 2 (ditto), and so on. Each node will be working single-mindedly on one block at a time. The current block that a node is trying to figure out is called a height. Every message that a node sends will be annotated with the block number that it's currently working on, and nodes will ignore all messages that concern any past orfuture blocks (with one small exception, explained below).
Looking ahead, one challenge will be that, in the asynchronous phase, different nodesmay be participating in different invocations of the single-shot protocol—some nodes maybe working to figure out block 9 (having already figured out the first eight blocks) whileother nodes, to whom key messages have been massively delayed, may still be working onan already-decided-upon block like block 7 so one node's height is nine and another's height is 9. When notes communicate to each other in Tendermint, every single message will be annotated with what block that node is trying to figure out. So, if we're trying to figure outblock number nine, every message we send to anybody will rather report the fact that we're trying to figure out block number nine. Now each note will be single-mindedly focused on the block it's currently stuck on, so if I'm working on number nine, We're going to ignore any messages we received from other nodes about block number seven which we already figured out.I'm going to ignore any messages from other nodes about block number 11 which we don't want to worry about yet. There's an exception: basically, if we're working on block nine, we listen to the messages that are also about block nine and we will ignore everything else, so in effect, the different single-shot consensus instances really don't interact with each other at all.

7.2.2 Idea 2: Aggressive Restarts

To explain the second idea, zoom in on a particular block, say block 9. As in the Byzantine broadcast problem and its application in SMR protocols (see chapter 2), there will be a "leader" node responsible for proposing a block to other nodes. There are two obvious issues that could interfere with honest nodes reaching agreement about block 9. The first is that the leader could be a Byzantine node, sending inconsistent information to different honest nodes. The second is that, before the global stabilization time, honest nodes' messages could be massively

delayed. The solution is for the nodes working on block 9 to restart (with the next leader) after a short period of time if they don't see sufficient progress. The hope (which we'll formalize in Section 7.5) is that eventually once GST has passed and the current leader is honest, if not earlier, the honest nodes will be able to come to agreement on block 9 and move on to block 10. One key challenge is that, due to inconsistent behavior by Byzantine nodes and/or big message delays, some but not all honest nodes may "see sufficient progress" by the restart time, and this should raise red flags about possible consistency violations. This issue is addressed by the last high-level idea.

7.2.3 Idea 3: Two Stages of Voting

The third and most clever idea in the Tendermint protocol is to use two stages of voting in each attempt by honest nodes to come to agreement on the next block. Why do we need two stages of voting rather than one? The issue, mentioned above, is that different honest nodes may perceive different outcomes of a vote. For instance, for a simple majority vote with the two options "commit to block B" and "try again with a new leader", some honest nodes may see over 50% of the votes for the first option and the other honest nodes over 50% supporting the second option. Again, each of two separate forces is powerful enough to cause different honest nodes to see different vote counts: inconsistent messages from Byzantine nodes (sending votes for one option to some honest nodes and the other option to the rest) and (at least pre-GST) big message delays (with some honest votes arriving on time and some delayed indefinitely).
Presumably, nodes that perceive the vote as a success will perform one action (e.g., commit the proposed block as block 9 to their local history) while the others perform a different action (e.g., restart with the next leader). This could easily lead to a violation of consistency(e.g., if the next leader proposes a different block for block 9 and the as-yet-uncommitted nodes agree to it).
Fundamentally, the problem with one stage of voting is that there are only two possible outcomes, and so disagreeing honest nodes perceive "diametrically opposite" outcomes.With two stages of voting—and with the second-stage vote taken only if the first-stage votesucceeds—there are three possible outcomes (first stage fails, first stage succeeds but second stage fails, and both stages succeed). While it's still the case that different honest nodes may observe different outcomes, the hope is that the "intermediate" outcome allows an honest node to hedge its bets between the more extreme outcomes of committing to a block and restarting the agreement process with a brand-new block. Intuitively, if an honest node observes a first-stage voting success and a second-stage failure, it will "lock in" on a block that it's pretty sure will wind up being block 9, but without irrevocably committing to it. The node remains open to subsequent arguments that it should lock in on a different block instead, but will only do so if confronted with overwhelming evidence that it is behind the times and needs to catch up with the most recent information. So that's the reason for two stages of voting; it's not clear two should be enough, but it gives the protocol more flexibility by having this third intermediate outcome between the restart and the commit outcomes.

7.3 The Tendermint Protocol

7.3.1 Preliminaries

Fidelity of description. There are many variants of the Tendermint protocol. The variant described in this chapter is meant to be the most straightforward possible implementation that is faithful to the three big ideas in Tendermint (Section 4.2), subject to actually being correct. This choice means sacrificing some amount of efficiency—for example, we won't worry about the details of who sends which messages to whom (every message will be signed by its sender and broadcast to everybody) or about optimizing constant factors in the protocol's rate of block production. (Of course, these details can be important for an efficient concrete implementation of Tendermint.) Another consequence is that the proofs of consistency and liveness (in Sections 4.4 and 4.5, respectively) will not be as slick as for some other variants.

PKI assumption. The Tendermint protocol makes use of the PKI assumption and begins with a commonly known list of nodes and their public keys. (Each node is assumed to know its private key. and each message is signed by the sender) The PKI assumption is not necessary to achieve optimal fault tolerance in the partially synchronous model (as originally shown by Dwork, Lynch, andStockmeyer), but it is a convenient and reasonably natural assumption in a blockchain setting.

Rounds. In the partially synchronous model (see Section 4.1), all the nodes know and agree on at all times the current time step (with no communication needed), even in the asynchronous phase. Also, there is a known upper bound Δ on the maximum number of time steps for which message might get delayed during the synchronous phase (after GST). In Tendermint, a round corresponds to 4Δ consecutive time steps and represents one attempt at reaching agreement on a block before restarting (see Section 4.2.2). The first round starts at time 0 and ends at time 4Δ, the second starts at time 4Δ and ends at time 8Δ, and so on. Because all nodes always know the current time step (and the length of each round), all nodes always know the current round number (even in the asynchronous phase). The protocol description depends on the value of the parameter Δ. This is allowed because the value Δ is assumed to be known a priori. (By contrast, the protocol description can not depend on the (unknown) global stabilization time).

Rotating leaders. Each round will have a unique leader node responsible for proposing a block, with nodes taking turns as the leader. This plan should remind you of our SMR protocol in Chapter 2 for the synchronous model, with each leader acting as the sender in a Byzantine broadcast subroutines such as the Dolev-Strong protocol. (In this chapter, without the benefit of synchrony, the Tendermint protocol can't afford to wait for a round to conclude with agreement on a block and instead restarts after a timeout.) Because the names of the nodes running the protocol and the current round are common knowledge (as we're in the permissioned model with a shared global clock), all nodes always know who the current leader is (e.g., node 1 in the first round, node 2 in the second round, and so on).

7.3.2 Quorum Certificates (QCs)

This section highlights a crucial definition (which is common to most "BFT-type" consensus protocols, not just Tendermint).

Votes. As suggested in Section 4.2.3, nodes will be voting on blocks. Such a vote has five attributes:

Note:-

Attributes of a Vote

1. the identity of the voter (as proved via a cryptographic signature that could only have been created by that node);
2. the block (i.e., ordered sequence of not-yet-executed transactions) that the vote is for;
3. the block number (e.g., block 9);
4. the round number (e.g., round 117);
5. the voting stage (first or second).

Accordingly, you can think of a vote as a 5-tuple (i, B, h, r, s). (in here "h" stands for "height," which is synonymous with the block number.) The first two attributes should be self-explanatory—votes are for specific blocks, and in order to avoid ballot-stuffing, the protocol must be able to associate each vote with a specific node. (Note that we're taking advantage of the PKI assumption here.) Section 3 foreshadowed the other three attributes. Because different instances of single-shot consensus shouldn't interfere with each other and different nodes may be working on different blocks (Section 4.2.1), votes should be annotated with the block number. Because reaching an agreement on a given block may require multiple restarts (Section 4.2.2), a vote must be explicitly associated with the round in which it was casted. Finally, because each attempt at agreement (in a given round, for a given block) requires two stages of voting (Section 4.2.3), a vote must indicate which stage it belongs to.

Definition 7.3.1: Definition of QCs

Call a triple (h, r, s) a referendum (on the outcome of stage s of voting within round r and for block number h). Here's the key definition (where as usual, n denotes the number of nodes, which is common knowledge in the permissioned setting):

Definition 7.3.2: Quorum Certificate (QC)

A quorum certificate (QC) is a set of votes from at least $\frac{2}{3}n$ distinct voters that are all for the same block in the same referendum.

In other words, all the votes in a QC agree in their last four components (and take on at least $\frac{2}{3}n$ different values in the first component). A QC represents a supermajority of support for a specific block (in a specific referendum). We will sometimes say that the QC supports that block. We can interpret a QC as a "successful vote" (for the supported block in the given referendum).

Proposition 7.3.1 Properties of QCs

Next is a simple but important lemma about QCs, which will also mark the first entrance of the famous "33%."

Lenma 7.3.1 QC Overlap Property

Every pair of QCs overlaps in at least $\frac{n}{3}$ nodes.

Proof: Let Q_1, Q_2 denote a pair of QCs. By definition, at most $\frac{n}{3}$ nodes are not represented in Q_1. Similarly, at most $\frac{n}{3}$ nodes are not represented in Q_2. This leaves at least $n-(\frac{n}{3})-(\frac{n}{3}) = \frac{n}{3}$ nodes that must be represented in both Q_1 and Q_2. 😊

Lemma 7.3.1 is true no matter how many nodes are Byzantine (all it is is some simple counting). But something special happens once the fraction of Byzantine nodes drops below one-third:

Corollary 7.3.1 Two QCs Overlap in an Honest Node

If $f < \frac{n}{3}$, then every pair of QCs overlap in at least one honest node.

How is this helpful? As we'll see, honest nodes in the Tendermint protocol will be instructed to vote at most once in each referendum. Byzantine nodes may engage in double voting (sending votes for a block B to some honest nodes and for a different block B to the rest), but with $f < \frac{n}{3}$, they will be unable to create QCs for two different blocks in the same referendum.

Corollary 7.3.2 At Most One QC Per Referendum

Suppose that every honest node votes at most once per referendum and that $f < \frac{n}{3}$. Then, if Q_1 and Q_2 are QCs for the same referendum, Q_1 and Q_2 support the same block.

Proof: By the previous corollary, some honest node i is represented in both QCs. By assumption, that node does not vote more than once in the referendum. Thus, both QCs must support the same block (whichever block node i voted for in that referendum). 😊

Newer and older QCs. In the Tendermint protocol, every (honest) node i will maintain two local variables, a block B_i and a QC Q_i that supports B_i. (One exception: when a node first starts working on a new block number, it sets Q_i to null and B_i to the as-yet unexecuted transactions that it knows about, ordered arbitrarily.) Intuitively, B_i indicates node i's current belief about what the next block should be. As we'll see, a node may cast aside an old QC in favor of a new one (for the same block number). Precisely, a (non-null) QC Q_1 with referendum (h, r_1, s_1) is more recent than another (non-null) QC Q_2 with referendum (h, r_2, s_2) if: (i) Q_1 is from a later round (i.e., $r_1 > r_2$); or (ii) Q_1, Q_2 are from the same round but Q_1 is from a later stage (i.e., $r_1 = r_2$ and $s_1 > s_2$). Any (non-null) QC is considered more recent than a null value. As mentioned, conceptually, the different single-shot consensus instances are not going to be interfering with each other right. So a block working on block number nine never worries about messages on any of the blocks. There is an exception to that, so if you're working on block number nine, you're going to ignore messages from previous blocks; those are already decided. Furthermore, you won't care anymore for future blocks, you're going to ignore a lot of things, but if your nodes are ever kind of circulating quorum certificates for a future block like block number 11, you will remember

those and you rather keep those in your back pocket. They're not going to be relevant to you until you catch up and get to block number 11, but you want those quorum certificates from the future block saved locally.

7.3.3 Protocol Pseudocode

As mentioned earlier, each round of the Tendermint protocol consumes 4Δ time steps, where Δ denotes the maximum message delay following the global stabilization time. Every (honest) node will take actions only at time steps that are multiples of Δ (with messages possibly received in between consecutive such multiples). Accordingly, each round has 4 phases, which we'll explain first in pseudocode and then in English. The pseudocode below is for a node i working on a particular block number h_i. The node ignores all messages received that are about different block numbers (with an exception, detailed at the end of this section). Also, remember that every sent message should be signed by the sender. Messages that could not have plausibly been sent by an honest node are automatically ignored by honest nodes. This applies, for example, to messages missing a signature; round-r block proposals by any node that is not the leader of round r; and any message beyond what is expected (e.g., if a node hears more than one block proposal from the same leader, it ignores all but the first).

Note:-

(A Version of) the Tendermint Protocol

Assumptions: local node i is working on block number h_i, with local variables B_i and Q_i (initialized as in Section 7.3.2). All messages for other block numbers are ignored (with one exception, see "in the background" below). Current round is r. Leader of round r is ℓ.

// First phase (executed at time $t = 4\Delta r$)
1 if $i = \ell$ then // local node is the current leader
2 if ℓ has received a height- h_i QC more recent than (B_ℓ, Q_ℓ) then
3 $B_\ell := B_j, Q_\ell := Q_j$ // update accordingly, to the most recent one (B_j, Q_j)
4 broadcast (B_ℓ, Q_ℓ) to all nodes // annotated with h_i, r, signature

---

// Second phase (executed at time $t = 4\Delta r + \Delta$)
5 if i has received (B_ℓ, Q_ℓ) from ℓ then // must be signed by ℓ
6 if Q_ℓ at least as recent as Q_i then // out-of-date, need to update
7 $B_i := B_\ell, Q_i := Q_\ell$
8 broadcast (B_i, Q_i) // keep all nodes up-to-date
9 Broadcast first stage vote for B_i // annotated with h_i, r, signature
// (no vote cast if (B_ℓ, Q_ℓ) not received on time)

// Third phase (executed at time $t = 4\Delta r + 2\Delta$)
10 if i has received at least $\frac{2}{3}n$ round- r (first-stage) votes for a block B then
11 $B_i := B$ // might or might not change the value of B_i
12 Q_i := the votes above // constitute a round-r stage-1 QC
13 broadcast (B_i, Q_i) // keep all nodes up-to-date
14 broadcast second-stage vote for B_i // annotated with h_i, r, signature
// (no vote cast if no round-r stage-1 QC is received on time)

// Fourth phase (executed at time $t = 4\Delta r + 3\Delta$)
15 if i has received at least $\frac{2}{3}n$ round- r second-stage votes for a block B then
16 $B_i := B$ // might or might not change the value of B_i
17 Q_i := the votes above // constitute a round $-r$ stage-2 QC
18 broadcast (B_i, Q_i) // keep all nodes up-to-date
19 commit B_i to local history as block number h_i // worry: consistency?
20 increment h_i // next round, will start working on the next block
21 reset B_i to the known as-yet-unexecuted transactions (ordered arbitrarily)
22 reset Q_i to null
// (no block committed if no round-r stage-2 QC received on time)

// Addendum (at time $t = 4\Delta r + 4\Delta$, just before first phase of round $r + 1$)

// (Catch up with all future blocks that have already been decided upon)
23 while i is in possession of a height- h_i stage-2 QC Q for a block B do
24 carry out lines 18-22 from the fourth phase (with (B, Q) playing the role of (B_i, Q_i))

// In the background (at all times)
25 store all QCs received for blocks $h_i + 1, h_i + 2, \ldots$ // for use in lines 2 and $23 - -24$

The round and leader are common knowledge. Let's walk through the pseudocode that dictates the behavior of the honest nodes. Consider an honest node i that is trying to figure out block number h_i (having already finalized and committed to blocks $1, 2, \cdots, h_{i-1}$), and consider the beginning of a round r. (Which is time step $4\Delta r$, assuming that we start numbering rounds at 0.) Because we're working in the permissioned and partially synchronous setting (with a shared global clock), and assuming a commonly known leader rotation (e.g., round robin in order of the node ID), node i automatically knows the current round r and the identity of the leader l of this round.

First phase. The first phase is relevant only if node i happens to be the round-r leader l. In this case, the node is responsible for proposing a block to the other nodes (much like the sender in the Byzantine broadcast problem). Node i first checks if it has received any (height-h_i) QCs from other nodes that are more recent than its incumbent QC, and if so, it updates its locally stored QC Q_i to the most recent one received (and the variable B_i to the block supported by Q_i). node i then broadcasts a proposal for the block B_i (along with the supporting QC Q_i) to all nodes. (For ease of exposition, imagine that node i also sends such a message to itself, which arrives immediately.) node i adds its signature to each of these messages, along with the block number h_i and the round number r.

Second phase. The second phase is meant for non-leaders to process any block proposals that might have been sent in the first phase. Each node i listens during the time steps $4\Delta r, 4\Delta r + 1, ..., 4\Delta r + \Delta$ for a block-QC pair sent by the leader l of the current round. (If more than one is received, all but the first are ignored. Proposals by nodes other than l are likewise ignored.) If no such pair is received by time step $4\Delta r + \Delta$ (due to a Byzantine leader or delayed messages), node i does nothing in the second phase. If node i receives such a pair (B_l, Q_l) in which Q_l is at least as recent as its locally stored QC Q_i, the node jettisons its values of B_i and Q_i and replaces them with the newly received values B_l and Q_l. Node i does this whether or not the new block B_l matches the old one B_i. In this case, the node also broadcasts the new values of (B_i, Q_i) to all nodes (to help them stay up-to-date), along with its first-stage vote for B_i. (If Q_i is more recent than Q_l, node i concludes that the leader l must be out of date (or Byzantine), and so it sticks to its guns with the incumbent values of (B_i, Q_i) and skips this phase's referendum.)

Third phase. In the third phase, nodes try to ascertain the outcome of the referendum that took place in the second phase of the round (without any bias from whatever vote they cast in that referendum). If a node receives a super-majority with at least two-thirds of the nodes represented, possibly including that node itself of round-r stage-1 votes for the same block B by the end of the time step $4\Delta r + 2\Delta$, it considers B the conclusive winner of the referendum. (By Corollary 7.3.2, assuming $f < \frac{n}{3}$, there is at most one such winner.) Node i adopts block B as its locally stored block B_i and assembles the received votes into a QC supporting B. (By definition, these votes constitute a QC, seen first-hand, for block B in the referendum at the previous phase.) This is the most recent QC imaginable, so node i adopts it as the new value of Q_i. Finally, it notifies all the nodes by broadcasting (B_i, Q_i), along with a second-stage vote for B_i (annotated as usual with the block number h_i , the round number r, the stage number $s = 2$, and i's signature). If node i does not receive such a supermajority by the end of the time step $4\Delta r + 2\Delta$, it does not cast a second-stage vote.

Fourth phase. In the fourth phase, nodes look for evidence of a successful referendum in the third phase (again, without any bias from whether they participated in that referendum or not). If a node receives a supermajority of round-r stage-2 votes for the same block B by the end of the time step $4\Delta r + 3\Delta$, it considers B the winner of that referendum. (Again, assuming $f < \frac{n}{3}$, there is at most one such winner. If node i does not receive such a super majority by the end of the time step $4\Delta r + 3\Delta$, it does nothing in this phase.) In this case, analogous to the third phase, node i adopts block B as its locally stored block B_i , assembles the received votes into a (seen first-hand and most recent possible) stage-2 QC supporting B, sets Q_i to this QC, and broadcasts (B_i, Q_i) to all the nodes. But now that B has survived not one but two stages of voting, node i takes the plunge and permanently commits block B as block number h_i in its local history. (Our proof of consistency in Section 5 will

have to rule out the possibility of any other honest node ever committing any other block as block number h_i in its history, even during the asynchronous phase.) Having committed to block number h_i, node i will switch to start working on block $h_i + 1$ in round $r + 1$, with its local variables B_i and Q_i reset appropriately (to the as-yet-unexecuted transactions the node knows about and null, respectively).

Final details. An honest node i working on block number h_i ignores messages about earlier blocks. It also ignores messages about future blocks, with one exception: whenever it receives a QC for a future block $h > h_i$ (as would be broadcast by an honest node in lines 4, 8, 13, 18, or 24), it saves it in its back pocket for future use. First, these saved QCs potentially come in handy the next time i is a leader (they would be processed in lines 2–3). Second, at the end of each round, node i processes any saved stage-2 QCs that it has received for future blocks, subject to the constraint that it commits to a height-h block only after committing to blocks for all previous heights. (There's no point in participating after-the-fact in the corresponding single-shot consensus instances, so node i skips them and directly adopts the already-agreed-upon future blocks.)

Perspective. It's not obvious that the Tendermint protocol satisfies consistency or eventual liveness when less than a third of the nodes are Byzantine, and you should demand careful proofs of both those properties (as will be supplied in Sections 7.6.4 and 7.6.5). Frankly, it's hard to imagine coming up with this protocol without simultaneously writing out the proofs of its key properties (a great example of how protocol analysis can inform protocol design). As we've said multiple times, intuition doesn't get us very far with distributed protocols. The devil is usually in the details, and getting those details right generally requires careful mathematical analysis.

7.4 Proof of Consistency

The goal of this section is to prove that the Tendermint protocol satisfies consistency, even in the asynchronous phase. The proof here sacrifices some degree of brevity in exchange for (relatively) easy line-by-line verifiability.

> **Theorem 7.4.1** Consistency of the Tendermint Protocol
>
> In the Tendermint protocol, if $f < \frac{n}{3}$ and two honest nodes commit blocks B and B' to their local histories as the same block number h, then $B = B'$.

Proof: We'll prove the theorem block by block, so zoom in on your favorite block number h. The plan is to prove that, if Q_1 and Q_2 are height-h stage-2 QCs (possibly from different rounds) supporting blocks B_1 and B_2, then $B_1 = B_2$. In other words, at no point in the protocol will there ever be two stage-2 QCs for a given block number that support different blocks. Because possession of a height-h stage-2 QC for a block B is a prerequisite for an honest node to commit to block B at the height h (as per the fourth phase of the pseudocode), this will imply that no two honest nodes ever commit to different blocks at height h. 😊

Consistency rules out the epic fail of two honest nodes committing to two different versions of the same block number. In other words, as soon as a single honest node commits a block B to its local history as block number h, that block can be considered as "finalized"— no other honest node will ever consider any block other than B as the rightful occupant of block number h.
Theorem 6.4.1 does leave open the possibility that some honest nodes will be lagging behind others. The theorem promises that the tortoises, once they catch up to the hares, will reach exactly the same conclusions.

Let r denote the first round in which more than $\frac{n}{3}$ honest nodes contribute height-h stage-2 votes in support of a common block; denote this set of honest nodes by S and the block they voted for by B^*. (Because a QC requires votes from at least $\frac{2}{3}n$ distinct nodes and $f < \frac{n}{3}$, this event is a prerequisite for the creation of a height-h stage-2 QC. That is, no such QC could have possibly been produced at any round prior to r.) The basic idea is that the nodes of S, having seen at least one successful vote for B^*, will remain "locked in" on B^* (refusing to cast votes for other blocks) until confronted with convincing evidence that they're in the wrong; and because S is so big and a QC requires participation by so many nodes, such evidence can never be created.

Formally, we can complete the proof by showing that there will never be a height-h stage-2 QC that supports a block other than B^*. We proceed inductively. For the base case, there cannot be a QC for the referendum $(h, r, 2)$ that supports a block $B \neq B^*$—because $|S| > \frac{n}{3}$ and at least $\frac{2}{3}n$ nodes must contribute to a QC, such a QC would

require participation from some (honest) node of S. That node would have then voted twice (once for B^* and once for B) in the same referendum $(h, r, 2)$. But as is evident from the protocol pseudocode, every honest node votes at most once in each referendum. To continue the argument, let's take stock of where things stand at the end of round r and beginning of round $r+1$ (at time step $4\Delta(r+1)$):

1. Under voting for the block B^* in the second stage of round-r voting (in line 14), every node of S was, at the start of time step $4\Delta r + 2$, in possession of a QC from the referendum $(h, r, 1)$ that supports the block B^*. (As usual, Corollary 7.3.2 then implies that every QC from this referendum must support B^*.)

2. At that same time, each such node $i \in S$ must have set its local variable B_i to B^* (and Q_i to its QC from referendum $(h, r, 1)$ that supports B^*); see lines 11 and 12.

3. By the base case argument, any QC that can be assembled from votes in the referendum $(h, r, 2)$ must support block B^*. Thus, no QC from this referendum can cause a node $i \in S$ to assign in line 16 its local variable B_i to anything other than B^*. (Such a node may or may not wind up updating Q_i in line 17 from a QC from referendum $(h, r, 1)$ to one from $(h, r, 2)$; if it does, it will wind up committing the block B^* to its local history in round r.)

Summarizing, at time $4\Delta(r+1)$: (i) for each $i \in S$ that has not already committed block B^*, $B_i = B^*$; (ii) for each i S that has not already committed block B^*, Q_i is a QC supporting B^* from referendum $(h, r, 1)$ or later; (iii) every height-h QC from a referendum $(h, r, 1)$ or later supports the block B^*.

Moving onto round $r+1$, properties (i)–(iii) imply that no node of S will change its mind about supporting B^*. (This is obvious for nodes that have already committed B^* to their local histories, as they will never again vote in any height-h referendum.) For example, in the first phase, if $l \in S$, the "if" statement in line 2 cannot be satisfied with any QC that does not support B^*, so B_l remains equal to B^* after this phase. Similarly, in the second phase, if $i \in S$, the "if" statement in line 6 cannot be satisfied by any QC that does not support B^*, and B_i remains equal to B^* after this phase. No such node will vote for a block other than B^* in the second phase, and so, because $|S| > \frac{n}{3}$, the referendum $(h, r+1, 1)$ cannot produce a QC for any block other than B^*. Consequently, no node of S will vote for a block other than B^* in the third phase (the "if" statement in line 10 cannot be satisfied for any block other than B^*), and the referendum $(h, r+1, 2)$ also produces no QCs for blocks other than B^*. Similarly, in the fourth phase, because of the "if" statement in line 15 cannot be satisfied for any block other than B^*, every node $i \in S$ (that hasn't already committed B^*) concludes the phase with $B_i = B^*$. Thus, each of the four phases preserves properties (i)–(iii).
Given that (i)–(iii) hold at the beginning of round $r+2$, history will repeat itself, with the same argument implying that (i)–(iii) also hold at the end of the round. By induction on the number of rounds, properties (i)–(iii) hold forevermore. By property (iii), and because round r is the earliest round at which a height-h stage-2 QC could be produced, we can conclude that there will never be a height-h stage-2 QC for a block other than B^*. This implies that no honest node will ever commit to a block at height h other than B^* , completing the proof.

7.5 Proof of Liveness

Let's move on to the second property that we care about, eventual liveness. (If all we wanted was consistency, we could easily use an SMR protocol that never does anything!) Here "eventual" means that liveness should hold soon after the network resumes "normal operating conditions" (i.e., soon after the global stabilization time, one of the two key parameters in the partially synchronous model that we reviewed in Section 7.1). The proof of Tendermint's consistency (Theorem 7.4.1) is not trivial, but it is relatively robust to variations in the protocol description—it relies primarily on the two stages of quorum certificates. The proof of Tendermint's liveness is more delicate and it depends heavily on some of the finer details in the protocol description.

7.5.1 Defining Liveness

In chapter 2, we defined liveness for the SMR problem (with clients submitting transactions to the nodes running the protocol) as:

Note:-

Liveness (Strong Version)

If a transaction is known to at least one honest node, that transaction is eventually added to every honest node's local history.

This property doesn't hold for the version of the Tendermint protocol described in this chapter—if a transaction tx is known to only one honest node i, it is possible that i will never get any of its block proposals finalized and thus tx is effectively censored, and never added to any node's local history. (Proving this fact is an excellent test for checking if you thoroughly understand the Tendermint protocol.) We'll work instead with a slightly weaker definition of liveness.

Note:-

Liveness (Weak Version)

If a transaction is known to all honest nodes, that transaction is eventually added to every honest node's local history.

We'd rather satisfy the stronger requirement than the weaker one, but I don't mean to make too big of a deal of the distinction. You can imagine the honest nodes exchanging transactions that they've heard about via a gossip protocol (thereby reducing the strong version to the weak version), or directly modifying the Tendermint protocol so that it (eventually) satisfies the stronger liveness property.

7.5.2 Fast Forwarding Past Obvious Obstructions to Progress

Fast forwarding past the GST. At what point can we hope for guaranteed progress, with honest nodes continually adding new blocks to their local histories? In the partially synchronous model, one obvious obstruction to progress is message delays. It's possible that no message ever gets delivered before the global stabilization time, so presumably, we'll need to fast-forward past the GST to guarantee liveness.

Fast forwarding past Byzantine leaders. A second obvious obstruction to liveness is Byzantine leaders—when a Byzantine node is a round's leader, it can give the other nodes the silent treatment (making no block proposals) to ensure that the round is wasted. So presumably, we'll need to fast forward to a round with an honest leader (and also after GST). This isn't enough—we need to fast forward to a pair of consecutive rounds that both have honest leaders. How can we be sure that there will ever be such a pair of consecutive rounds? Assume that leaders rotate in some round-robin order (with each node acting as the leader every n rounds). For there to never be a pair of consecutive honest leaders, there would need to be a Byzantine leader at least every other round. With round-robin leader rotation, this can happen only if at least 50% of the nodes are Byzantine. Our standing assumption is that less than one-third of the nodes are Byzantine, so we're good to go:

Lenma 7.5.1 The inevitability of Consecutive Honest Leaders

Assuming round-robin leader election and that $f < \frac{n}{3}$, there are infinitely many pairs of consecutive rounds with honest leaders.

7.5.3 The Proof

Here's the claimed liveness guarantee for the Tendermint protocol:

Theorem 7.5.1 Liveness of the Tendermint Protocol

In the Tendermint protocol with round-robin leader rotation, if $f < \frac{n}{3}$ and a transaction tx is at some time step known to all honest nodes, that transaction is eventually added to every honest node's local history.

To start the proof, suppose a transaction tx is known at time step t to every honest node. Fast forward to the first pair of consecutive rounds r_1, r_2 with honest leaders such that round r_1 begins at or after time step $\max\{t, GST + \Delta\}$. (Lemma 7.5.1 assures us that such a pair exists.) Let l_1 and l_2 denote the (honest) leaders of rounds r_1 and r_2, respectively.

The rough idea is that the first honest leader l_1 will synchronize all honest nodes—in effect, flushing the system of any foolishness that previous Byzantine leaders might have been up to—and the second honest leader l_2 will lead them to commit a block that is guaranteed to contain tx (assuming that tx has not already been included in some previous block). There are a lot of details, however, and going through them should give you a visceral feel for how tricky consensus protocol design really is.
First, a lemma:

> **Lenma 7.5.2** Near-Convergence of Local Block Numbers
>
> Every honest node begins round r_1 working on block number h or h + 1, where h denotes the highest block number that any honest node is working on at time $t^* := 4\Delta r_1 \Delta$.

Proof: 😊

Consider the fourth phase of the round r0 that immediately proceeds r_1, which takes place at time $4\Delta r_0 + 3\Delta = t^*$. By assumption, $t^* \geqslant GST$. At time t^*, no node (honest or otherwise) possesses a QC for a block number higher than h (as any QC requires some votes from honest nodes, and no honest node has yet voted for any block number higher than h). Suppose honest node i is working on block number h at time t^*. This node must, at that time, possess stage-2 QCs for blocks $1, 2, \cdots, h1$. All of these QCs would have been echoed to all nodes at time t^* if not earlier (in line 24 of some earlier round, or in one of lines 4, 8, or 18 of the current or some earlier round). Because $t^* \geqslant GST$, all these echoed QCs are received by all honest nodes by the end of round r_0. After the processing of these QCs in line 24 of round r_0, all honest nodes begin round r_1 working on block number h or higher. Because there are not yet any QCs for any block numbers higher than h, every honest node must begin round r_1 working on block number h or $h + 1$.

Unfortunately, a Byzantine node can force some honest nodes to work on block number h in round r_1 and others to work on block number $h + 1$; the idea is to keep a height-h stage-2 QC secret until time t^* and then release it selectively to some but not all honest nodes. (Thinking this Byzantine strategy through is a good exercise.) We address this complication via a case analysis.

Case 1: Everyone's Working on Block Number $h + 1$ Suppose every honest node begins round r_1 working on block number $h + 1$. Because there are not yet any QCs for block numbers higher than h, all honest nodes begin the round with a null QC. Events then unfold as follows (using repeatedly that round r_1 is post-GST and hence all messages arrive within Δ time steps):

1. In the first phase of round r_1, the (honest) leader l_1 proposes a block B that includes all not-yet-executed transactions that the leader knows about. If the target transaction tx has not already been included in some previous block, it will then be included in B.

2. In the second phase, all honest nodes receive the leader's proposal of block B (and so the protocol proceeds to line 6). Because there are not yet any QCs for block number $h+1$, the "if" statement in line 6 is satisfied and all honest nodes proceed to execute lines 7–9.22 Thus all honest nodes issue a first-stage vote for block B in line 9.

3. In the third phase, all honest nodes receive each other's first-stage votes for B. Because $f < \frac{n}{3}$, these votes constitute a stage-1 QC and all honest nodes proceed to execute lines 11–14. In particular, all honest nodes issue a second-stage vote for block B in line 14.

4. In the fourth phase, all honest nodes receive each other's second-stage votes for B. These constitute a stage-2 QC and all honest nodes proceed to execute lines 16–22. In particular, all honest nodes commit to block B as block number $h+1$ in line 19. Because the target transaction tx is included in either B or some previous block, the proof for case 1 is complete.

In general, call a round clean if: (i) the round begins at or after GST; (ii) the round has an honest leader; (iii) all honest nodes begin the round working on the same block number h; and (iv) after the processing in lines 2–3, the (height-h) QC Q_l stored locally at the leader l is at least as recent as every QC Q_i stored locally by an honest node at the beginning of the round. The four-step argument above proves:

Lenma 7.5.3 Clean Rounds Commit Proposed Blocks

Every clean round concludes with all honest nodes committing to their local histories (as block number h) the block proposed by the leader in the first phase.

In particular, condition (iv) ensures that, in the second phase, the "if" condition in line 6 is met at all honest nodes, who then issue first-stage votes for the block proposed to them by the (honest) leader in the first phase.

Case 2: Everyone's Working on Block Number h Suppose every honest node begins round r_1 working on block number h. In particular, no honest node has received a height-h stage-2 QC by the start of this round (at time $t^* + \Delta$). We claim that this round is clean. Because properties (i)–(iii) hold by assumption, we only need to verify (iv).

Let (B, Q) denote the most recent height-h (and stage-1) QC known to any honest node at time t^* and the block that this QC supports. (If there are no such QCs, define Q to be null and B to be the non-yet-executed transactions that the leader l_1 knows about.) Analogous to the proof of Lemma 7.5.3, this block-QC pair will have been echoed and received by the round-r_1 leader l_1 by the beginning of that round. Thus, in the first phase of round r_1, after the processing of lines 2–3, the (honest) leader l_1 has a locally stored height-h block-QC pair (B^*, Q^*) for which Q^* at least as recent as Q.
Now consider an honest non-leader i. If i was working on a block number less than h in round r_0 (only catching up to the others in the fourth phase or line 24 of that round), it begins round r_1 with the null QC (which is no more recent than Q^*). If i was working on block number h at round r_0, the value of its locally stored QC Q_i at the beginning of round r_1 is the same as it was at the time t^* (and is therefore, no more recent than Q). Either way, the value of Q^* (after the processing of lines 2–3) is at least as recent Q_i (at the beginning of round r_1), which verifies (iv). Knowing that round r_1 is clean, Lemma 7.5.4 implies that all honest nodes commit block B^* to their local histories as block number h. We're not done, however: because B^* may be an inherited block proposal from an earlier round, possibly even from a Byzantine leader, there is no guarantee that it includes the target transaction tx. But the next round r_2 is also clean, with all honest nodes working on block number $h+1$ and with null QCs. (Remember, no height-$(h+1)$ QCs exist yet.) By Lemma 7.5.4, the round concludes with all honest nodes committing to the block B proposed by the (honest) leader l_2 as block number $h+1$. Because B was proposed by an honest node in a time step after tx was known to all such nodes, it is guaranteed to include tx (assuming that tx isn't already in some previously confirmed block).

Case 3: The Leader Lags Behind Suppose that the leader l_1 begins the round $_1$ working on block number h, and at least one honest node i begins the round working on block number $h+1$. Node i must then have in its possession a height-h stage-2 QC Q^* for a block B^* at the beginning of the round. Because (B^*, Q^*) was echoed at or before this time (in one of lines 4, 8, 18, or 24 of an earlier round), all honest nodes have received this height-h stage-2 QC by the end of round r_1. Thus, by the end of round r_1 (in line 24 if not earlier), every honest node will have committed to some height-h block. Thus, round r_2 will be clean, with all honest nodes working on block number $h+1$ and with null QCs. (There are not yet any height-$(h+1)$ QCs: Round r_1 was the first in which any honest node would have been willing to vote in a height-$(h+1)$ referendum, but the block proposed in that round by l_1 was for block number h.) As in case 2, Lemma 7.5.4 implies that the target transaction tx will be included in some block (either the one produced in round r_2 or some earlier block).

Case 4: The Leader Is Ahead Finally, suppose that the leader l_1 begins the round r_1 working on block number $h+1$ and that at least one honest node begins the round working on block number h. Assume that the target transaction tx has not been included in any of the first h blocks (as otherwise, we're done). The block B^* proposed by the (honest) leader l_1 (for block number $h+1$) in the first phase of this round will contain tx. At the start of round r_1, no height-$(h+1)$ QCs exist. Any such QCs produced in round r_1 must support the block B^* . In particular, if any honest node commits to block number h+1 in round r_1, it must be to B^* . In this case, we're done. (Block B^* contains tx and all honest nodes will end up committing to block B^* , in line 24 of round r_2 if not earlier.) If no honest node commits to block number $h+1$ in round r_1, then all honest nodes begin round r_2 working on block number $h+1$. This round is clean, for the exact same reasons that round r_1 in case 2 is clean. By Lemma 7.5.4, round r_2 concludes with all honest nodes committing (as block number $h+1$) the block proposed by l_2 in the first phase of the round. This block is either the proposal B^* inherited from the previous round or a new block proposal B' assembled by l_2 in the first phase (which, because l_2 is honest, will contain tx).

Either way, the target transaction tx will be added to all nodes' local histories, completing the proof of liveness.

7.5.4 Chain Quality

Theorem 7.5.2 and its proof have interesting consequences for the "chain quality" of the sequence of finalized blocks, meaning the fraction of such blocks that are contributed by honest nodes. Call a post-GST round with an honest leader bad if the round does not conclude with all honest nodes committing a block that was originally proposed by an honest node. In all four cases of the proof of Theorem 7.5.2, the block committed by honest nodes in the second (clean) round r_2 is a block that was proposed by an honest node. Thus, every bad round must be preceded by a round with a Byzantine leader. Because more than two-thirds of the rounds have honest leaders and less than one-third have Byzantine leaders (assuming round-robin leader rotation), more than one-third of the post-GST rounds conclude with the commitment of an honestly assembled block. Because less than one-third of the post-GST rounds conclude with the commitment of a block assembled by a Byzantine node (as any given round can lead to at most one confirmed block), more than half of the blocks finalized post-GST were originally proposed by honest nodes. Using chain quality to mean the fraction of confirmed blocks proposed by honest nodes, we thus have:

Theorem 7.5.2 Chain Quality of the Tendermint Protocol

After the global stabilization time, the chain quality of the Tendermint protocol is more than 50More generally, if at most an $\alpha < \frac{1}{3}$ fraction of the nodes are Byzantine, the chain quality (post-GST) is at least

$$\frac{1-2\alpha}{1-\alpha}$$

We will see this expression again in the next chapter, in the context of longest-chain consensus.

7.6 Can We Do Better?

With the tricky proofs out of the way, let's take a step back and ask: Could we hope to do better than Tendermint? Here are a few senses in which the protocol is "optimal":

1. The upper bound on the number f of Byzantine nodes cannot be relaxed (without compromising elsewhere). We saw in Chapter 6 that when $f \geqslant \frac{n}{3}$, no consensus protocol satisfies consistency and eventual liveness in the partially synchronous model.

2. The assumption of partial synchrony cannot be relaxed to asynchrony (without compromising elsewhere). We studied the FLP impossibility theorem in chapters 4–5, which implies that in the asynchronous setting, and even with only one Byzantine node, no deterministic protocol can guarantee both consistency and eventual liveness.

3. Holding the bound on f and the model of communication model (i.e., partial synchrony) fixed, we cannot strengthen "eventual liveness" to "liveness" (without sacrificing consistency). This again follows from the FLP impossibility theorem—during the asynchronous phase, you cannot have both consistency and liveness.

So is Tendermint the end of the story in state machine replication? Is there any reason to talk about any other SMR protocols?

Alternate trade-offs. One reason to consider alternative protocols is to achieve different trade-offs—different points on the Pareto curve. While no protocol will strictly dominate Tendermint in all of the dimensions listed above, there could be interesting protocols that are stronger in some respects and weaker in others.
For example, we could stick with the partially synchronous model and ask: Can we strengthen eventual liveness to liveness (always), while relaxing consistency to eventual consistency? In other words, could we escape the FLP impossibility theorem by giving up on consistency rather than liveness during the asynchronous phase? (In the synchronous phase, as usual, we should expect to have both.) This was not a particularly popular question in the 20th-century literature on consensus protocols, but it is exactly the alternative trade-off that protocols like Bitcoin and (the original version of) Ethereum make.

More efficient protocols. A second active and practically important direction is to develop consensus protocols that match the guarantees of Tendermint (in terms of fault-tolerance, consistency, and liveness) but have better performance. "Performance" means different things to different people, but some example metrics include the number of messages sent per round, the duration of a round, the number of times nodes need to carry out certain expensive cryptographic operations per round, and so on.

As promised at the beginning of this boot camp on classical consensus protocols, we haven't paid much attention to any of these kinds of performance metrics. One reason is the lack of agreement among experts about which metrics are the most important. A second reason is that a detailed performance analysis would be a little out in the weeds for our purposes. This chapter series is fundamentally about blockchains, not consensus protocols per se, and there are a lot of other interesting aspects of blockchains that we want to have time for.

You should know, however, that there have been several advances in BFT-type protocols since Tendermint, and some of these appear likely to make it to production over the next few years. For example, Facebook/Meta's Diem project derived its consensus protocol from HotStuff, which from 30,000 feet can be viewed as a sort of pipelined, more efficient version of Tendermint. The Diem project has been dissolved, but its consensus protocol looks likely to resurface in other projects.

Another example is the Ethereum blockchain, which is slated to make a major upgrade in 2022 that would, in particular, incorporate some BFT-type ideas into its existing longest-chain consensus protocol. (We'll discuss basic longest-chain consensus in chapter 8.) For example, if you look into "Casper FFG"—a "finality gadget" responsible for finalizing periodic checkpoints of the underlying longest-chain protocol—you'll again see something that resembles (at a high level) a pipelined version of Tendermint. HotStuff and Casper-FFG have the same fault tolerance as Tendermint, and in particular, there is again the magical 33% threshold on the fraction of Byzantine nodes that can be tolerated (subject to consistency and eventual liveness in the partially synchronous model). Having survived this boot camp on consensus protocol basics (chapters 2–7), you are well positioned to understand these recent advances.

Optimistic responsiveness. While on the topic of efficiency, one bummer about the Tendermint protocol is that each round takes 4Δ time steps, even if all messages are being delivered super-quickly. (E.g., maybe Δ corresponds to 1 second but it's a good network day, with all messages delivered within 10 milliseconds.) Optimistic responsiveness is the property that, whenever the current round's leader is honest, the time required to confirm a block scales with the actual (rather than worst-case) message delay. This property is one of the claims to fame of the aforementioned HotStuff protocol, for example. Protocols with this property generally differ from Tendermint in that new leaders are chosen only on an as-needed basis (if nodes detect that something has gone wrong with the current leader), rather than automatically in round-robin order. The idea of as-needed-leader-election comes originally from the famous PBFT (for "practical BFT") consensus protocol; in that context, a switch in the leader is known as a "view change." Tendermint can be viewed as (pessimistically) enacting a view change in every round.

Onward to 21st-century consensus protocols, and the longest-chain consensus made famous by the Bitcoin protocol!

Chapter 8

Longest-Chain Consensus

8.1 A Tale of Two Protocol Paradigms

Chapter 7 was about the Tendermint protocol (for state machine replication), and its optimal fault-tolerance in the partially synchronous model (with consistency always and liveness eventually). This was the culmination of our six-chapter boot camp on classical consensus protocols. Most of what we talked about was figured out by brilliant computer scientists in the 1980s. (Tendermint is a 21st-century protocol, but heavily influenced by the classic protocols from the 1980s and 1990s.1) As you may know, the most famous blockchain protocol of them all (Bitcoin) is based on a consensus protocol that does not at all resemble those that we've been discussing thus far. In the blockchain world, there are currently two prevalent paradigms for consensus protocols, "BFT-type" and "longest-chain" protocols.

8.1.1 Category 1: BFT-type Protocols (e.g., Tendermint)

Protocols like Tendermint are often called BFT or BFT-type protocols; the "BFT" stands for "Byzantine fault tolerant." The first shared property of these protocols is that they all look similar from 30,000 feet, with multiple stages of voting and some analog of quorum certificates. This ensures that a node commits to a new block only after it can be certain that (assuming not too many Byzantine nodes) no other honest node will ever commit to any other version of that block.

By design, under appropriate assumptions (such as $f < \frac{n}{3}$, where f and n denote the number of Byzantine and total nodes, respectively), BFT protocols do not suffer from "forks"—there will never be two different blocks committed by honest nodes at the same block height. If there ever is a fork in a BFT-type protocol (either due to a buggy implementation or more Byzantine nodes than expected), it's not at all clear how to resolve the fork from within the protocol.

Because BFT-type protocols favor consistency over liveness (when the network is poor or under attack), when they fail (for whatever reason), they typically fail by stalling (i.e., not confirming any new blocks of transactions for a prolonged period of time). You do not typically hear, for instance, about double-spend attacks caused by the rollback of thought-to-be-finalized blocks.

8.1.2 Category 2: Longest-Chain Protocols (e.g., Bitcoin)

If you've watched even a 20-minute video on Bitcoin at some point in your life, you already know there's a second category of blockchain consensus protocols, namely longest-chain protocols. The longest-chain consensus was invented by Nakamoto and first described in the Bitcoin white paper. Conceptually, longest-chain protocols are radically different from BFT-type protocols:

1. There is no explicit voting - each block (along with an explicit pointer to a predecessor block) is proposed unilaterally by some node.

2. Because there's no waiting for quorum certificates, forks - two (potentially conflicting) blocks that claim a common predecessor - can easily arise in the normal course of business (either due to Byzantine nodes or network delays). With potentially frequent forks, such a protocol can function only if it has an in-protocol way of automatically resolving the ambiguity introduced by a fork. This is done by - wait for it - always

resolving forks in favor of the longest chain of blocks. (We'll get into all the protocol details and nuances starting in the next section.)

3. Under embracing forks and resolving them in the protocol, longest-chain protocols favor liveness over consistency when there's a communication breakdown between nodes. Accordingly, in practice, longest-chain protocols tend to fail by violating consistency (with nodes disagreeing on which blocks have been finalized, leading to thought-tobe-confirmed blocks getting rolled back) rather than by stalling. [5] Such consistency violations can enable "double-spend" attacks. For example, suppose a blockchain transaction carries the payment for an expensive physical good like a Tesla. Once the Tesla dealer regards the transaction as confirmed, they let the buyer drive off in their new car. If that transaction later gets rolled back, the buyer effectively gets a full refund even though they still have the car.

8.2 Longest-Chain Consensus

8.2.1 Preamble

The version of the longest-chain consensus described in this chapter may differ from what you had in mind.

This chapter: permissioned setting, PKI assumption. Throughout this chapter, we will continue working in the permissioned setting with the PKI assumption (with an a priori known set of nodes running the protocol, each with a private key and a known-to-all corresponding public key). One reason for this is the continuity of all of the chapters thus far. Another reason is that many of the novel ideas in and properties of longest-chain consensus manifest already in this setting.

That said, the longest-chain consensus's biggest claim to fame is its shockingly graceful extension to the "permissionless" setting in which the protocol has no idea what nodes are running it. The next chapter (Chapter 9) is all about permissionless longest-chain consensus (specifically, the "proof-of-work" version used in Bitcoin).

Looking ahead toward the Bitcoin protocol. As I've said many times, this chapter series is focused on principles of blockchain design and analysis rather than on specific blockchain protocols per se. And if you study Bitcoin for Bitcoin's sake, for example, you generally wind up conflating multiple independent innovations. We'll take care to unbundle these innovations in this chapter series, both because of the mental clarity that it affords and because the separate pieces can be remixed with other ideas to design other interesting blockchain protocols. A consensus protocol lays down the law about which of the proposed blocks are the ones that count (and about their ordering). A conceptually distinct question is about which nodes can participate (e.g., by proposing a new block or voting on a previously proposed block). It's easy to come up with answers to this question in the permissioned setting (with the PKI assumption) - for example, by letting all nodes vote and having nodes take round-*r*obin turns acting as the block proposer. It's much harder in the permissionless setting because the protocol has no idea which nodes are running it. The second big innovation in Bitcoin (in addition to longest-chain consensus) is an effective way of selecting block proposers in longest-chain consensus in the permissionless setting, using an idea known as "proof-of-work."
In this chapter series, we'll keep these two questions (which blocks count? Who participates in block production?) as separate as possible. They will interact a little bit at their edges, though, as we'll see (see page 13).

Permissioned + PKI vs. proof-of-work vs. proof-of-stake implementations. Some readers may enter this chapter with the mindset of permissioned consensus (e.g., those coming in directly from chapters 2-7) while others may be strongly influenced by famous permissionless blockchain protocols that they've read about (e.g., Bitcoin and Ethereum). I'm going to try to have my cake and eat it too, by addressing both audiences at the same time.

The plan is for the main narrative of this chapter to focus squarely on longest-chain consensus in the permissioned setting (with the PKI assumption) but with frequent side comments and forward pointers about how to interpret that narrative in the content of permissionless blockchain protocols. We'll provide interpretations both for the proof-of-work implementation described in chapter 9, and the "proof-of-stake" implementation covered in chapter 12. Don't worry about it if you've never heard about approaches to "sybil-resistance" like proof-of-work or proof-of-stake - the main narrative of this chapter assumes nothing other than familiarity with permissioned protocols for state machine replication (as covered in the preceding chapters).

Remember the Dolev-Strong protocol? Another reason to keep looking forward toward permissionless implementations is that in the setting of this chapter (permissioned setting with the PKI assumption), the longest-chain consensus is strictly dominated by an SMR protocol that we analyzed way back in Chapter 2 - the (iterated) Dolev-Strong protocol. That

[6] The combination of longest-chain consensus with proof-of-work block proposer selection, as introduced in Bitcoin, is sometimes called "Nakamoto consensus." protocol has some impressive provable guarantees - consistency and liveness even when 99% of the nodes are Byzantine-provided you're willing to make a bunch of assumptions: the permissioned setting, the PKI assumption, and the synchronous model (with an a priori known bound Δ on the maximum message delay).

As we'll see (Section 8.11), longest-chain consensus loses consistency in the partially synchronous setting and for this reason is studied primarily in the synchronous setting. Here, the protocol can guarantee consistency and liveness provided at most 49% of the nodes are Byzantine (see Theorems ?? and 10.1). It's not trivial to design a protocol with such guarantees (nor is it easy to prove that longest-chain consensus does, in fact, offer them), but one has to concede that - in the permissioned setting, with the PKI assumption, in the synchronous model - there's really no reason to use longest-chain consensus instead of simply running the Dolev-Strong protocol once for each block (with rotating or randomly chosen block proposers/senders).

Thus, it's only once we pass to the permissionless setting that the longest-chain consensus allows us to reach goals that we don't otherwise know how to achieve. This chapter is, therefore, best viewed as a stepping stone toward the proof-of-work and proof-of-stake implementations of longest-chain consensus studied in chapters 9 and 12, rather than as an end unto itself. But I promise that I'm not wasting your time - the key points in this chapter are all essential even for the reader who cares only about understanding permissionless longest-chain consensus.

8.2.2 Protocol Description

Our description of longest-chain consensus has three scenarios simultaneously in mind:

(PKI) Permissioned setting, PKI assumption.
(PoW) Permissionless setting, proof-of-work sybil-resistance.
(PoS) Permissionless setting, proof-of-stake sybil-resistance.

If you're unfamiliar with any permissionless blockchain protocols, ignore all comments about scenarios (PoW) and (PoS) - all will become clear in chapters 9 and 12, respectively.

An underspecified version. Let's start with an abstract and underspecified description of the longest-chain consensus, before interpreting it in each of the three scenarios above.

Note:-

Longest-Chain Consensus (Abstract Version)

(1) Start with a hard-coded genesis block B_{gen} .
(2) In each "round" $r = 1, 2, 3, \ldots$:

(2a) Choose one node ℓ as the leader of round r.
(2b) Node ℓ proposes a set of blocks, each specifying a single predecessor block.

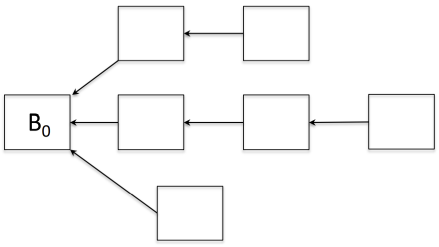


Figure 8.1: The blocks produced in longest-chain consensus can be visualized as a tree, directed toward the genesis block (root).

We'll supply more details below, but already at this level of specification we can fruitfully visualize the data structure produced by the nodes running the protocol as an in-tree - a tree (in the sense of an acyclic connected graph) in which all edges are directed back toward the root (Figure 8.1). For us, the genesis block acts as the tree's root, and each directed edge corresponds to one block's naming of its predecessor. [7] When the protocol is first fired up, this in-tree consists solely of the genesis block; as the rounds go by, nodes running the protocol continue to grow this in-tree (with each leader adding a new batch of vertices, each with out-degree 1). Note that, as promised, forks (a block named as the predecessor of multiple blocks, or equivalently a vertex with in-degree greater than 1) are embraced as part of normal operation.

The genesis block. Every longest-chain protocol has baked into it a genesis block that gets the party started. There are no transactions in the genesis block, but it is an eligible predecessor for any block that might be created later. Just as each node is born knowing the protocol code (and, in the PKI setting, its private key and all nodes' public keys), each node should be thought of as born knowing the genesis block.

What's a "round"? Each iteration of longest-chain consensus, each with a single leader node who's granted the power to add to the current blockchain is called a round. The meaning of a round depends on which scenario we're talking about. In scenarios (PKI) and (PoS), we'll assume a shared global clock (as in the synchronous model) and each round will correspond to a time interval of a fixed length. For example, round 1 might be the first 10 seconds, round 2 the next 10 sections, and so on. We've seen this idea before (in scenario (PKI)), for instance, in the iterated Dolev-Strong protocol in chapter 2 and the Tendermint protocol in chapter 7.
Interestingly, in scenario (PoW), rounds will not correspond to fixed periods of time (except in a loose average sense), and more generally, there's only minimal reliance on a shared notion of time. Rather, rounds are defined in a purely event-driven way. Readers who know how proof-of-work works - with block producers racing to be the first to solve a difficult cryptographic puzzle - may already see what I mean by "event-driven." Every time some block producer gets lucky and solves a puzzle (the event of interest), we'll call that the next round.

How is the leader chosen? Each round has a single node that acts as the leader. The abstract description above is silent on how this happens in step (2a), and the answer depends on which scenario we're looking at.
The simplest scenario is (PKI). We could reuse our approach from chapters 2 and 7 of round-$r$obin rotating leaders. E.g., if there are 100 nodes and it's currently round 117, node 17 is the leader. Alternatively, and better for longest-chain consensus (as we'll see), leaders could be chosen (pseudo)randomly - for example, using a hash $h(117)$ of the current time step rather than the time step 117 itself to identify the current leader. Either way, assuming that each round has a fixed length and that all nodes share a global clock, they all automatically know what round it is, and in the permissioned setting they all then know which node is the leader at any given time.
You can, at a high level, think of scenario (PoS) as similar to the second approach in scenario (PKI), with leaders chosen randomly. A twist is that leaders will be not be selected uniformly at random, but rather with probability proportional to the amount of stake (in the blockchain's native currency) that a node has locked up in a designated smart contract. E.g., if 10% of the total stake locked up in the contract is owned by node i, then in any given round, node i has a 1 in 10 chance of being chosen as the round's leader. (We'll talk about all the nuances of implementing this idea in chapter 12.) Scenario (PoW) is similar to (PoS) except that a node is effectively selected as a leadermeaning it's the first node to solve a hard cryptographic puzzle - with probability proportional to the amount of computational power that it contributes to the protocol (rather than to the amount of locked-up stake).

What can the leader do? Once a node is selected as the leader of a round (in step (2a)), in step (2b) that node gets to create some number of blocks. Here a block is, as usual, an ordered sequence of transactions (which were presumably submitted to nodes earlier by the clients participating in the SMR protocol). However many blocks that leader creates, it must specify a unique predecessor block for each of those blocks. (Any block that specifies zero or more than two valid predecessors is automatically considered invalid by all honest nodes. Looking ahead, an honest leader will be instructed to create exactly one block; it's Byzantine nodes that might create several.) The leader is always in a position to do this - if nothing else, it can use the (commonly known) genesis block as a predecessor.

As we'll see below, exactly what the leader is capable of doing will depend on which of the scenarios we're in (e.g., the leader is most highly constrained in scenario (PoW)) - this is the part of longest-chain consensus in which there is inevitably interaction between the details of the consensus protocol and the method by which nodes are selected to participate.

The use of explicit predecessors is a departure from BFT-type protocols like Tendermint in which, assuming

less than a third of the nodes are Byzantine, there will be only one block at each block height (because quorum certificates are required for block finalization, and by the quorum intersection property). Because there's only going to be one block #8, one block #9, one block #10, and so on, there's no need to explicitly name a predecessor-everyone knows that the predecessor is the (unique) block at the previous height. In longest-chain consensus, any leader can create a fork if it wants (e.g., by proposing multiple blocks, all with the same predecessor). Thus there may be multiple blocks at each block height, effectively competing for eventual finalization. This renders explicit predecessors necessary - if there are multiple blocks at height 8, a block at height 9 needs to specify which one it views as the preceding block.

8.3 Five Assumptions

The goal of this chapter is to show that longest-chain consensus satisfies consistency and liveness in the synchronous model provided at most 49% of the nodes are Byzantine. We'll carry out this analysis under five assumptions, detailed and discussed next.

8.3.1 Assumptions About the Genesis Block

The first assumption is a trusted setup assumption - meaning an assumption that we take on faith, kicking the can down the road as to how it might be enforced. Specifically, we assume that no nodes are privy in advance to the description of the protocol's genesis block-nodes only learn the name of that block at the moment the protocol commences. (E.g., the genesis block can't have been chosen by some Byzantine node in advance.)

Note:-

Assumption (A1)

(A1) No node has knowledge of the genesis block prior to the deployment of the protocol.

As a trusted setup assumption, we'll make no effort to enforce it within the protocol itself (regardless of which of the three scenarios we're in). When deploying a longest-chain protocol, a good faith gesture is to include a genesis block that references verifiable information that could not have realistically been predicted well in advance of the protocol's deployment. This assumption will show up in an important (but subtle) way in our proof of Theorem 8.7.1.

8.3.2 Assumptions About Leader Selection

We'll also need two assumptions about how leader selection works in step (2a) of longestchain consensus. These are not trusted setup assumptions, and we'll need to make sure that a concrete implementation of longest-chain consensus enforces them. [9]

Note:-

Assumption (A2)

(A2) It is easy for all nodes to verify whether a given node is the leader of a given round.

The second assumption asserts that each round's leader selection should be verifiable. This means two things: the round's selected leader can easily prove it is in fact the leader; and no other node can trick honest nodes into thinking it's actually the leader.

Happily, assumption (A2) takes care of itself in all the scenarios that we're interested in. In scenario (PKI), imagine that leaders rotate in round-robin order, e.g. with node 17 automatically the leader of round 117 (assuming $n = 100$). Imagine further that every message sent by an (honest) node is signed by the sender and annotated with the round it belongs to. Because of the PKI assumption, node 17 and only node 17 will be in a position to send messages as the leader in round 117 (if a round-117 message that is supposed to be from the round's leader is not signed appropriately, all honest nodes ignore the message). The same reasoning holds for pseudorandomly chosen leaders.

In scenario (PoS), leader selection will typically be done via a deterministic function whose output (the public key of the leader) can be easily verified. (There will effectively be a pseudo-PKI assumption when we study

scenario (PoS) in chapter 12, so again the current leader and only the current leader will be in a position to send messages that will be accepted by the honest nodes as coming from the round's leader.)

In scenario (PoW), as we'll see in chapter 9, leadership is literally defined by exhibiting a proof of it (in the form of a solution to a difficult cryptographic puzzle) in tandem with the choice of block and predecessor in step (2b). In effect, the way longest-chain consensus works in scenario (PoW), every message comes equipped with a self-contained proof that the node was authorized to send it.

We also need a second assumption about leader selection:

Note:-

Assumption (A3)

(A3) No node can influence the probability with which it is selected as the leader of a round in step (2a).

Intuitively, it would seem problematic if a Byzantine node could concoct a strategy that resulted in getting selected as the leader of every round. Ideally, leader selection should be completely unmanipulatable by the nodes running the protocol—in effect, with the choice of a leader just falling from the sky in each round.

Once again, in scenario (PKI) this assumption is easy to implement (with rotating or pseudorandomly chosen leaders). Every node automatically knows the current round and therefore (by the permissioned assumption) the current round's leader—there's nothing anyone can do about it.

Enforcing assumption (A3) becomes non-trivial in the permissionless case, though in scenario (PoW) things work out surprisingly simply. We'll see in chapter 9 that, under something known as the "random oracle assumption" about cryptographic hash functions, there exist cryptographic puzzles such that all practical purposes can be solved only by repeated guessing (with each guess independent and equally likely to produce a solution). Under this assumption, you could think of all nodes as repeatedly throwing darts at a dartboard, attempting to hit the (really small) bullseye and thereby becoming the next round's leader. It's hopefully intuitive that a node's likelihood of being the first to hit the bullseye is proportional to its number of attempts (i.e., its computational power) and is otherwise unaffected by anything else that the node might do.

Enforcing assumption (A3) in scenario (PoS) is tricky and we'll talk about it at length in Chapter 12. Intuitively, the difference between scenarios (PoW) and (PoS) is that, in the former, there is a natural external source of randomness (generated by repeated dart-throwing). In the latter scenario, it would seem that the protocol must generate (pseudo)randomness on its own, based only on the information available in its hermetically sealed environment (i.e., the protocol code and the sequence of transactions executed thus far) and without the benefit of external randomness. This should sound dangerous—e.g., maybe a node can influence its future probability of selection by manipulating the state of the blockchain. Over the past several years, designers of proof-of-stake blockchains have been continually coming up with new ideas to better enforce assumption (A3)—more on this in chapter 12.

Assumptions (A2) and (A3) are both crucial whenever we argue (e.g., in Section 8.6) that the assumption of at most 49% Byzantine nodes (or computational power in scenario (PoW) or stake in scenario (PoS)) translates to an analogous assumption about the fraction of leaders selected in step (2a) that are Byzantine.

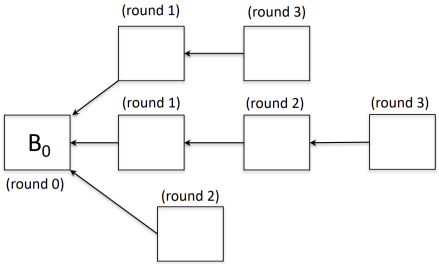


Figure 8.2: Blocks with round numbers consistent with an assumption (A4): tracing predecessor pointers back from a block produce a sequence of blocks with strictly decreasing round numbers.

8.3.3 Assumptions About Block Production

The next assumption plays a fundamental role in our analysis of longest-chain consensus (e.g., in the proofs of Theorem 8.7.1 and 8.9.1), and we'll need to take care to enforce it in any concrete implementation.

Note:-

Assumption (A4)

(A4) Every block produced by the round-r leader must claim as its predecessor some block that belongs to a previous round.

In other words, if you trace predecessor pointers back from a block, you will encounter a sequence of blocks with strictly decreasing round numbers (ending in the genesis block, which we interpret as belonging to round 0). In particular, because this assumption rules out any cycles among the produced blocks, our visualization of longest-chain consensus as an evergrowing in-tree is accurate. This assumption also automatically rules out the possibility of two blocks from the same round appearing on a common chain (Figure 8.2).

Your first reaction to assumption (A4) might be that it seems obvious—how can a block refer to something that doesn't exist yet? The assumption rules out two possibilities that would otherwise be problematic. First, remember that in step (2b) a (Byzantine) node might create multiple round-r blocks; assumption (A4) makes the non-trivial assertion that none of these round-r blocks can point to each other as predecessors. (The node can still create a bunch of round-r blocks, for example using some round-$(r-1)$ block as the predecessor for all of them.) Second, one possible strategy for a Byzantine node that is selected as the round-r leader is to delay the announcement of its choices in step (2b) until later. (We inevitably have to deal with this possibility, because it can be indistinguishable from an honest node that has had its messages delayed.) For example, imagine node 17 is a Byzantine node selected as the leader of round 117, and only in round 125 does it announce "by the way, here are my round-117 blocks and their predecessors," with some of the named predecessors created only in rounds 118–124. Assumption (A4) asserts that Byzantine nodes should be unable to get away with this.

Assumption (A4) is crucial to our analysis (as will be obvious in Sections 8–11), but it's actually pretty trivial to enforce in all three scenarios that we're thinking about. In scenario (PKI), for example, the obvious approach is to require that each block proposal is accompanied by a round number and a signature from the leader of that round, and for honest nodes to ignore any block proposal in which the round number of the predecessor block is at least as large as that of the new block. The exact same idea can be used in scenario (PoS).

A stronger property in the proof-of-work setting. Scenario (PoW) is the interesting one, and in fact blocks will not be explicitly annotated with their round number. As we'll see in chapter 9, typical proof-of-work-based implementations of longest-chain consensus require nodes to commit to their choices in step (2b) before they're even notified that they're the round's leader. (Technically, solutions to cryptographic puzzles are regarded as valid by honest nodes only if they include an encoding of the node's step (2b) choices, and validity can only be checked after the proposed solution has been assembled.) Because they commit to their block proposals before round r has started (it starts only once they've validated their winning lottery ticket, including the block proposals encoded by the ticket), those blocks can only refer to predecessor blocks that exist at that time (which, by definition, were created in rounds prior to r). Given that, in scenario (PoW), winning lottery tickets by definition must encode the node's decisions in step (2b), why not further restrict the ticket format so that the node specifies exactly one (rather than many) round-r block, along with its predecessor? (As we'll see, honest nodes are anyways supposed to propose exactly one block in a round of longest-chain consensus.) By incorporating this idea, typical implementations of longest-chain consensus in scenario (PoW) wind up enforcing a stronger version of assumption (A4):

Note:-

Assumption (A4) [Stronger Proof-of-Work Version]

(A4') The leader of a round produces exactly one block, and this block claims as its predecessor some block that belongs to a previous round.

The fundamental consistency and liveness guarantees of longest-chain consensus (Theorems ?? and 10.1) depend only on assumption (A4), not on the stronger version satisfied by typical proof-of-work instantiations. (Though, as we'll see, the proof of consistency does get easier if you can assume (A4') and not just (A4).) This is exactly the kind of insight that you miss by studying the Bitcoin protocol for its own sake rather than by

examining each of its ingredients in isolation. This point also demonstrates the value of crisply articulating the minimal assumptions required for the validity of an argument—as it turns out, many ideas that were originally developed to analyze the (PoW) scenario specifically can be reused directly for the other two scenarios as well.

8.3.4 Assumptions About Communication

You should be bothered by our final assumption, which is patently false:

Note:-

Assumption (A5)

(A5) At all times, all honest nodes know about the exact same set of blocks (and predecessors).

Alternatively, you can think of assumption (A5) as asserting that we're working in the synchronous model with the maximum message delay Δ equal to zero—whenever, an honest node learns anything new, it can then instantly communicate (by clairvoyance, in effect) the new information to all the honest nodes. We'll sometimes call this the "super-synchronous" or "instant communication" model (neither of which are standard terms).
Even when we work in the synchronous model with $\Delta = 1$, there will be periods of time in which different honest nodes know different things. For example, when an honest leader proposes a new block, it knows about that block one time step earlier than the other honest nodes.

The plan. Unlike assumption (A1), we can't take (A5) on faith—it's simply not true. Unlike assumptions (A2)–(A4), we can't design a consensus protocol to enforce it—message delays are outside the control of the protocol. So what's the plan and the reasoning behind it?

1. We'll adopt assumption (A5) temporarily, and will relax it later. (We could relax it in this chapter, but the most sensible place to do so is in chapter 9, in the context of scenario (PoW).)

2. The vast majority of interesting and non-trivial ideas in the analysis of longest-chain consensus are necessary even under assumption (A5).

3. These ideas are easier to understand in the safe confines provided by assumption (A5), unoccluded by other details.

4. All the guarantees we'll prove for longest-chain consensus under assumption (A5) will hold more generally in the traditional synchronous model, provided the maximum message delay Δ is small relative to the average length of a round.

5. The analysis of this extension to the synchronous model (discussed in chapter 9) is conceptually straightforward once the main ideas in this chapter (under assumption (A5)) have been properly internalized. (Warning: some amount of math is involved.)

Consistency vs. finality. Still, you'd be right to ask: "Isn't the whole point of state machine replication to keep nodes in sync? And haven't you trivialized the problem by asserting that all nodes automatically know the same information?" That's a good criticism. Assumption (A5) doesn't trivialize liveness, but it does seem to trivialize consistency. But let's split "consistency" into two conceptually different properties:
(i) consistency between different nodes at a given moment in time; and
(ii) "self-consistency," meaning consistency between an honest node and a future version of itself.

Property (i) is obviously trivialized by assumption (A5), so let's look at property (ii). "Consistency with one's future self" means if an honest node i regards a block B as finalized at time t, then the block should never be rolled back: node i should regard B as finalized also at all moments in time after t. (The order of finalized blocks should also remain the same.) We'll sometimes refer to this property as—wait for it—finality. In previous chapters, property (ii) was baked into the definition of the SMR problem— remember that every node maintains an append-only data structure of transactions. (Note "append-only" is equivalent to "nothing gets rolled back.") And for BFT-type consensus protocols like Tendermint, the way we proved consistency for different nodes—by arguing via quorum certificates that there will never be two different blocks finalized at the same block height—immediately implies that (assuming $f < \frac{n}{3}$) no honest node would ever roll back an already-committed block. Longest-chain consensus, however, can be vulnerable to violations of finality in the form of rollbacks. Even

under assumption (A5), it is not at all trivial to prove that the protocol satisfies such self-consistency under reasonable assumptions. And happily, the main ideas that prove finality (ii) under assumption (A5) (in Theorem 8.8.1) can be largely reused to also prove consistency between nodes (i) when assumption (A5) is relaxed (to the traditional synchronous model, with a parameter Δ that is small relative to the average round length).

8.3.5 Recap

Here are the five assumptions, collected together for convenience:

Note:-

Assumptions (A1)–(A5)

(A1) No node knows the genesis block prior to the deployment of the protocol.
(A2) It is easy for all nodes to verify whether a given node is the leader of a given round.
(A3) No node can influence the probability with which it is selected as the leader of a round in step (2a).
(A4) Every block produced by the round-r leader must claim as its predecessor some block that belongs to a previous round.
(A5) At all times, all honest nodes know about the same set of blocks (and predecessors).

You should be wondering about the roles of these assumptions in this chapter's analysis. The analysis will be modular, with two main steps. One step identifies a sufficient condition on the sequence of leaders chosen in step (2a) of longest-chain consensus—a type of "balancedness property"—under which the protocol satisfies consistency and liveness (see Section 8–11). The other step (see Section 8.6) identifies conditions (on the amount of Byzantine participation and the method of leader selection, e.g. random leader selection) that guarantee the generation of a balanced leader sequence (perhaps with high probability). Assumptions (A2) and (A3) are about the integrity of the leader selection process; accordingly, they show up (somewhat implicitly) in Section 8.6, as part of proving that Byzantine leaders will not be overly frequent.
The other three assumptions are all important for ruling out the possibility that infrequent Byzantine leaders can cause an outsized amount of havoc on the protocol. The role of assumption (A1) is a bit subtle, while assumptions (A4) and (A5) will be crucial to the analysis. I think it will also be intuitively clear from this chapter's analysis that assumption (A5) is overkill and that similar arguments should work (perhaps with a slightly messier proof) in the general synchronous model (with delay parameter Δ well less than the typical round length). Here's a table summarizing the roles of the five assumptions:

Assumption	Type	Used in Which Proofs?
(A1)	trusted setup	Theorem 8.7.1
(A2)	to be enforced	implicit in Section 8.6
(A3)	to be enforced	implicit in Section 8.6
(A4)	to be enforced	Theorems 8.7.1, 8.9.1, and 8.10.1
(A5)	to be relaxed	Theorems 8.7.1, 8.9.1, and 8.10.1

8.4 Honest vs. Dishonest Behavior

8.4.1 Prescribed Behavior for Honest Nodes

Now that we understand what longest-chain consensus looks like at a high-level, let's discuss what honest nodes are supposed to do (in step (2b), once selected as a leader) and how Byzantine nodes might deviate from that prescribed behavior.

Honest blocks include all transactions. First, when selected as a leader, an honest node assembles a block from scratch as in our previous SMR protocols—by including all the as-yet-unexecuted transactions that it knows about (perhaps directly from a client, or perhaps from another node), in some arbitrary order. Note that an honest node proposes exactly one block each time it's selected as the leader of a round.

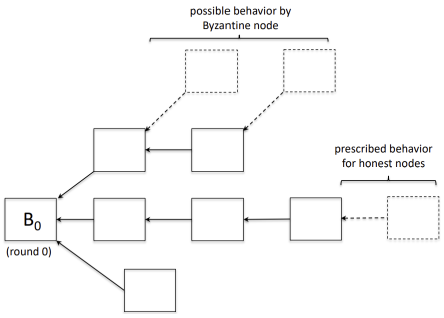


Figure 8.3: Honest nodes are instructed to propose a single block that extends the longest chain (with ties broken arbitrarily). Byzantine nodes might propose multiple blocks, and might name predecessors other than the end of the longest chain.

Extend the longest chain. What about the predecessor? An honest leader is instructed to look over the in-tree of all the blocks that it knows about and set its block's predecessor to the existing block that is furthest (in terms of number of hops) from the root (i.e., from the genesis block). In other words, an honest node adds its block to the end of the longest chain, making that chain still longer (Figure 8.3). And what if there's a tie, with two or more equally long chains? For our consistency and liveness analysis, we'll be pessimistic and assume that honest nodes break ties in an arbitrary (e.g., adversarially chosen) manner. You could imagine imposing a tie-breaking rule that honest nodes are supposed to follow (e.g., randomly, or in favor of the block heard about earlier), but guarantees as important as consistency and liveness should not be so fragile as to hinge on how honest nodes carry out tie-breaking.

Immediate announcements. Finally—this may seem obvious, but it is something that Byzantine nodes might deviate from—honest nodes are expected to broadcast their chosen block and predecessor immediately after they discover that they're the leader of the current round. Deviations by Byzantine Nodes A Byzantine node might deviate from the three instructions above in arbitrary ways— proposing more than one block or a block that doesn't contain all known transactions (or even an empty block), naming as a predecessor a block other than the end of the longest chain, and delaying the announcement of block proposals until some later point in time.

Empty blocks. The first type of deviation doesn't seem that bad—we've had to deal with the threat of Byzantine leaders proposing dishonest blocks ever since the SMR protocol based on iterating the Dolev-Strong protocol back in Chapter 2. Unlike in Chapter 2, however, there is no guarantee that any given honestly proposed block will eventually get finalized. Our liveness analysis in Section 8.9 will have to address this issue by arguing that honest nodes regularly manage to assemble blocks that eventually get finalized (if all finalized blocks came from Byzantine block proposers, they might all be empty in which case all transactions would be starved).

Delayed block announcements. For the consistency and liveness analysis in this chapter, the third type of deviation (delayed block announcements) turns out not to matter (other than making the proofs more annoying). Looking ahead, it will matter in Chapter 10 when we study the case of block producers competing for "block rewards." (As we'll see, this also relates to the "chain quality" discussion in Section 8.10.) We'll see in that chapter that, perhaps counterintuitively, there are scenarios in which a node can increase its block rewards (relative to honestly following the protocol) via a deviation that includes strategically delayed block announcements.

Deliberate forking. Deviations of the second type—the deliberate creation or perpetuation of forks through the extension of blocks other than the end of the longest chain—should be immediately worrying, and they will be the focus of this Chapter's analysis. Why are they worrying? Honest nodes are effectively trying to coordinate on a single (longest) chain, and absent interference by Byzantine nodes the longest chain would keep growing longer and longer. A block that does not extend the longest chain, not only fails to make the longest chain longer, but also threatens to switch which chain is the longest, which would lead to a rollback of blocks on the previously longest chain.
For example, suppose there are 10 rounds in a row with Byzantine leaders. These Byzantine nodes can collaborate and grow a fork of length 10 starting from 9 blocks back, to roll back the last 9 blocks of what had been the

longest chain (Figure 8.4). More generally, a sequence of k consecutive Byzantine leaders is in a position to roll back $k-1$ blocks. Still more generally, if there is a sequence of leaders in which the Byzantine leaders outnumber the honest leaders by k, the Byzantine nodes can roll back the last $k-1$ blocks of what had been the longest chain (why?). For example, if in a window of 1000 rounds, there are 510 Byzantine leaders and 490 honest leaders, the Byzantine nodes are in a position to roll back the last 19 blocks of what was the longest chain at the start of the window. (In fact, under our assumption of worst-case tie-breaking by honest nodes, such a set of Byzantine nodes could even roll back k blocks.)

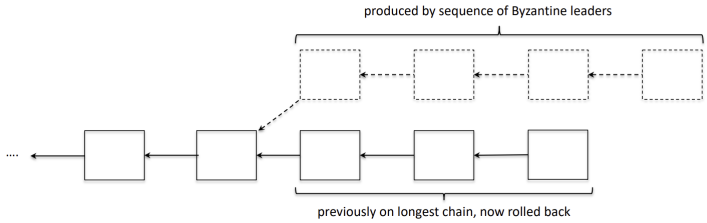


Figure 8.4: A sequence of k consecutive Byzantine leaders are in a position to roll back $k1$ blocks. (Or even, with worst-case tie-breaking by honest nodes, k blocks.)

Takeaways. What can we learn from these examples of deliberate forking? Three things:

1. With longest-chain consensus, you should always regard the last several blocks of the longest chain as tentative—not-yet-finalized and still under negotiation. In other words, a block should be considered finalized only once it is ensconced sufficiently deeply on the longest chain, already extended by several other blocks. How deep on the longest chain does the block need to be before it can be safely considered finalized? That's an important question, which we'll start tackling head-on in the next section.

2. The examples above point out a fundamental obstruction to proving any kind of finality result for longest-chain consensus: if you can't rule out sequences of leaders in which the number of Byzantine leaders outnumbers the number of honest leaders by k, then you can't expect any kind of finality to hold for any of the last $k-1$ (or even k) blocks on the longest chain. The best-case scenario would be that this is the only obstruction, meaning that the converse also holds: if you can rule out such windows, then finality does hold for all the blocks on the longest chain other than the last k of them. This converse constitutes the essence of our forthcoming analysis of the consistency of longest-chain consensus.

3. This style of deliberate forking shows that there's no hope of proving anything about longest-chain consensus unless strictly less than half of the nodes are Byzantine. To see this, imagine that 51% of the nodes are Byzantine. Assume that leader selection is done in a way that every node gets its fair share of turns. Then, 51% of the rounds will have Byzantine leaders. In a window of 1000 consecutive leaders you're likely to see something like 510 Byzantine nodes and 490 honest nodes, and these Byzantine nodes are then in a position to roll back the last 20 blocks of the once-longest chain. In a window of 10000 nodes, if there are 5100 Byzantine leaders and 4900 honest leaders, the Byzantine nodes will be able to roll back the last 200 blocks of the once-longest chain. And so on. Given that Byzantine leaders can outnumber honest leaders by an arbitrarily large amount (as the sequence length grows large), no block is ever safe from being rolled back—the Byzantine nodes effectively have dictatorial control over which blocks wind up on the longest chain. So, a necessary condition for the longest-chain consensus to have any provable guarantees is that more than half of the nodes run the protocol honestly and correctly; the best-case scenario would be that this condition is also sufficient (for provable consistency and liveness). Great news: this is exactly what we'll prove!

8.5 Which Blocks Are Finalized?

Given the power of the potential deviations (specifically, deliberate forking) by Byzantine nodes identified in the previous section, we've homed in on the coolest statement that might conceivably be true about longest-chain consensus: that whenever more than 50nodes running the protocol are honest, all blocks on the longest chain

other than the last k can be considered finalized. (Along with the other usual desired properties, like liveness.) And this is exactly the main punchline of the analysis in this chapter.

8.5.1 The Parameter k

The parameter k-the number of blocks at the end of the longest chain that are still considered unfinalized and under negotiation—will obviously be an important one for us. On the one hand, we'd like it to be as small as possible (so that blocks of transactions get finalized as soon as possible after they are created); on the other, intuitively, larger (more conservative) values of k seem more likely to allow for provable consistency guarantees. Let's introduce some notation to reason about this parameter: for an in-tree G of blocks (rooted at the genesis block) and a nonnegative integer k, define:

$$B_k(G) := \text{the longest chain of } G\text{, with the last } k \text{ blocks removed.} \quad (1)$$

For example, for the in-tree G in Figure 8.5, $B_2(G)$ is the chain $B_0 \leftarrow B_2 \leftarrow B_4$ while $B_3(G)$ is the chain $B_0 \leftarrow B_2$. There are two possible points of confusion about this definition.

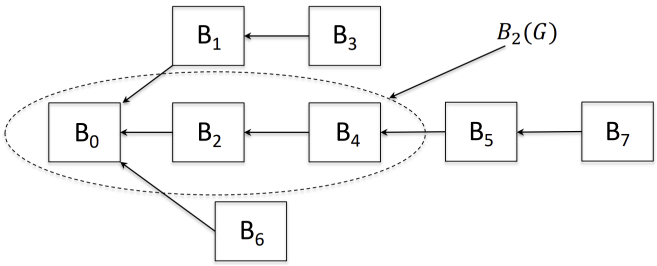


Figure 8.5: $B_k(G)$ denotes the longest chain of the in-tree G, with the last k blocks removed

Defined client-side rather than in-protocol. If you look at the description of longestchain consensus in Section 8.2, you'll notice that there's no parameter k—the evolution of the protocol is completely independent of what value of k you might have in mind. Rather, the parameter k dictates how to interpret the evolution of the protocol—how many blocks at the end of the longest chain you should regard as "still tentative" in some sense. The parameter k is in the eye of the beholder, and can therefore be set client-side, and differently for different clients. (We're using "client" here in the usual tech sense—the userfacing application that interacts with the blockchain—rather than in the strict SMR sense.) For example, imagine a blockchain protocol based on longest-chain consensus that also has a native currency (like Bitcoin), and imagine that you're a seller who accepts payments in this currency. If you're a high-volume seller of a low-cost product, like cups of coffee, you might be content to hand over a cup of coffee to a customer after that customer's payment transaction has been added to the longest chain and extended by a single block ($k = 1$). You'd run some risk of the transaction getting rolled back (and the customer effectively getting a free cup of coffee), but perhaps you view minimizing customer wait time as more important. If you're selling Teslas, on the other hand, probably you want to make the customer wait for awhile (maybe $k = 100$), so that you can be confident that their payment can be regarded as finalized, never to be evicted from the longest chain. Is $B_k(G)$ well defined? The second point of confusion, which you'd be right to wonder about, is whether the notation $B_k(G)$ is even well defined in a mathematical sense. In (1), who am I to write "the longest chain," when we all know that there might be multiple chains tied for the longest? Figure 8.6 depicts an in-tree G with two longest chains. Here, $B_2(G)$ actually is well defined—no matter which of the two longest chains you choose, after lopping off the last two blocks, you get the same thing (the chain $B_0 \leftarrow B_2$). But $B_1(G)$ is not well defined, because if you lop off only one block from the end, you're still left with different chains ($B_0 \leftarrow B_2 \leftarrow B_4$ and $B_0 \leftarrow B_2 \leftarrow B_5$). In general, when we say that "$B_k(G)$ is well defined," we mean that it doesn't matter which longest chain you choose, after lopping off the last k blocks you get the same thing. Equivalently, all longest chains agree on all their blocks except possibly for their last k. Said still another way, all longest chains share a common prefix—if they all have length l, then the prefix of the first $l - k$ blocks is shared across them.

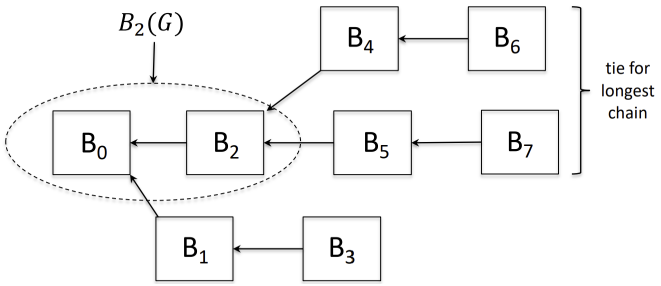


Figure 8.6: When there is a tie for the longest chain, $B_k(G)$ may or may not be well defined.

8.5.2 Balanced Leader Sequences

In our analysis, the first order of business (Theorem 8.7.1) will be to prove this "common prefix property": provided more than 50% of the nodes are honest and k is sufficiently large (and under our standing assumptions (A1)–(A5)), $B_k(G)$ is guaranteed to be well defined. From there, the goal will be to prove finality, meaning that the blocks of $B_k(G)$ can be considered finalized (Theorem 8.8.1), and liveness, meaning that $B_k(G)$ continues to grow over time, with blocks assembled by honest nodes regularly added (Theorem 8.9.1).

The plan. Our analysis will be modular. To explain, consider a sequence of rounds of longest-chain consensus and their corresponding leaders. If you think about it, there's no need to differentiate between which honest node is selected as a leader (each is following the protocol and thus behaving identically) nor which Byzantine node is selected as a leader (we always assume that Byzantine nodes are acting in cahoots, so it doesn't matter which one is chosen). We can therefore think of the leaders of a sequence of rounds as a bunch of "H"s (for honest) and "A"s (for adversarial). For example, we argued in the previous section that in any sequence of rounds in which there are k more "A"s than "H"s, the Byzantine nodes are in a position to cancel the last k blocks of the current longest chain. The plan for the modular analysis is then:

1. (Definition 8.5.1) State a definition that articulates a condition that a given leader sequence may or may not meet.

2. (Sections 8.7–8.10) Prove that, as long as the sequence of leaders generated in longestchain consensus satisfies this condition (and the parameter k is chosen appropriately, and assumptions (A1)–(A5) hold), the protocol satisfies all the properties that we want (consistency, liveness, etc.).

3. (Section 8.6) Prove that, as long as more than half of the nodes running the protocol are honest, this condition is satisfied (perhaps with high probability).

Chaining together the second and third steps shows exactly what we want: as long as more than half the nodes are honest and the parameter k is set appropriately, longest-chain consensus enjoys provable consistency and liveness guarantees.

Balanced leader sequences. Here's the key definition (with respect to a parameter w, which stands for "window" and is a positive integer):

Definition 8.5.1: w-Balanced Leader Sequence

A sequence $_1, l_2, l_3, \cdots \in \{H, A\}$ is wbalanced if, in every window $l_i, l_i + 1, \cdots, l_j - 1, l_j$ of length at least w, the H's outnumber the A's.

The bigger w is, the easier Definition 8.5.1 is to satisfy (because there are fewer windows to worry about). Thus we will generally be interested in the smallest w for which a leader sequence is w-balanced.
For example, any leader sequence that includes at least one Byzantine node fails to be 1- or 2-balanced (in a window of length 1 or 2 that contains that node, the honest nodes do not outnumber the Byzantine nodes). A sequence in which every pair of Byzantine leaders are separated by at least two honest nodes is 3-balanced (why?). A sequence generated by the round-robin rotation through n nodes, with $f < \frac{n}{3}$ Byzantine nodes, is n-balanced (why?). (Homework: what about with the weaker condition $f < \frac{n}{2}$?) A leader sequence in which the average

frequency of Byzantine nodes exceeds that of honest nodes—e.g., with $f > \frac{n}{2}$ and either round-robin or random leader selection—will not be w-balanced for any w (why?).

Implementing the second step of the plan. Let's connect Definition 8.5.1 back to the second takeaway from Section 8.4. We saw in that section how, in a window in which there are k more Byzantine nodes than honest nodes, Byzantine nodes can cancel the last k blocks of the current longest chain. The best-case scenario is that this obstruction to finality is the only obstruction to finality, meaning that as long as there are no windows in which the number of Byzantine nodes outnumber the honest nodes by more than k, all but the last k blocks of the longest chain can safely be considered finalized. In a w-balanced leader sequence, meanwhile, there cannot be any window in which the Byzantine nodes outnumber the honest nodes by $\frac{w}{2}$ or more (why?). The dream, then, would be that whenever Definition 6.1 is satisfied with some parameter w, all but the last $\frac{w}{2} - 1$ blocks of the longest chain can be considered finalized. And this is exactly what we're going to prove in Theorem 8.8.1 (under the assumptions (A1)–(A5) from Section 8.3)! Specifically, in Section 8.7 we'll use the w-balancedness condition to establish the common prefix property (Theorem 8.7.1)—i.e., that the notation $B_k(G)$ above is well defined, with all longest chains agreeing on all but possibly their final k blocks. The finality guarantee in Theorem 8.8.1 then follows easily in Section 8.8. We'll also use the w-balanced condition in the proofs of liveness (Theorem 8.9.1 in Section 8.9) and chain quality (Theorem 8.7.10.2 in Section 8.10).

Implementing the third step. The second step of our modular analysis shows that if your leader sequence is balanced—for whatever reason—then you're good, with all the consistency and liveness properties that you might want. But why should the leader sequence be balanced in the first place? We already argued that it won't be balanced if more than half the nodes are Byzantine, but why should it be balanced if more than half the nodes are honest?

In Section 8.6 we'll drill down on the version of longest-chain consensus in which each leader is chosen independently and uniformly at random. (This is a natural design already in scenario (PKI), but more importantly it's completely essential to the permissionless implementations of longest-chain consensus in chapters 9 and 12 for scenarios (PoW) and (PoS), respectively.) The punchline of the analysis in Section 8.6 will be that, as long as less than half the nodes are Byzantine, random leader selection generates a balanced leader sequence with high probability. The intuition is that—by the law of large numbers, essentially—honest nodes will have (almost) proportional representation in every sufficiently large window of consecutive leaders. (E.g., if 51% of the nodes are honest, then in all sufficiently large windows there should be somewhere between 50.5% and 51.5% honest nodes.)

How balanced a sequence, exactly—how big do the window lengths need to be before proportional representation kicks in? (Remember that the more balanced the leader sequence is—i.e., the smaller w is—the smaller we can take k and the faster blocks can be finalized.) The answer depends on various parameters (the fraction of nodes that are Byzantine, the duration you're interested in, and the acceptable failure probability), and we'll get into the details in the next section. For typical parameter values, the analysis suggests taking k (the number of additional blocks needed before considering a block to be final) in the low-to-mid double-digits. More aggressive values are often used in practice—for example, for the Bitcoin protocol the famous rule of thumb is to take $k = 6$. (Though when selling a Tesla, k should definitely be taken to something larger than that!)

8.6 Random Leaders Are Balanced

8.6.1 Random Leaders and Probabilistic Guarantees

We'll see in the next few sections proofs that, as long as the sequence of leaders generated in longest-chain consensus is w-balanced (see Definition 8.5.1), the protocol satisfies consistency and liveness (with the parameter k chosen appropriately). When can we count on a balanced sequence of leaders? And for what balance parameter w?

In this section we'll investigate, in scenario (PKI), the version of longest-chain consensus in which, every round, the leader is selected uniformly at random from the set of all nodes (i.e., with each node equally likely to be chosen). This is a natural approach to take in the permissioned setting, but it also extends amazingly gracefully to the permissionless setting (scenarios (PoW) and (PoW), see chapters 9 and 12).

Even in the permissioned setting, given that we want a balanced leader sequence, random leaders sound like a pretty good idea. For example, imagine that there are 3000 nodes, 1000 of which are Byzantine. If we cycle through the nodes in some fixed round-robin order, for all we know all 1000 Byzantine nodes appear consecutively

in the ordering. In this case, the leader sequence generated is 2001-balanced but not w-balanced for any $w < 2001$ (why?). If instead each leader is selected randomly, then any given leader has a two-out-ofthree chance of being honest. You might of course get a few Byzantine leaders in a row by random chance, but getting something like 100 Byzantine leaders in a row—an event with probability (1/3)100—would presumably happen so rarely that we could ignore the possibility.

Probabilistic finality and liveness. On the other hand, no matter how big w is, there is some positive—if astronomically small—probability of seeing w Byzantine leaders in a row. And this would be true even if there was only one Byzantine node! Thus with randomly selected leaders, there's no hope of proving any guarantees for longest-chain consensus that hold with certainty. (For example, any block might eventually get rolled back if the protocol winds up choosing a sufficiently long sequence of consecutive Byzantine leaders.) The bestcase scenario would be to prove that consistency and liveness hold with high probability (meaning probability close to 1, like 99.9%). Such probabilistic guarantees are typical of permissionless implementations of longest-chain consensus (including Bitcoin). If the failure probability is sufficiently small (e.g., less than the probability of your neighborhood getting hit by an asteroid in the next 24 hours), then probabilistic guarantees are for all practical purposes as good as deterministic guarantees.

8.6.2 Intuition for the Analysis

Taking the second step of this chapter's analysis (Section 8.7–8.10)—that a balanced leader sequence gives you strong consistency and liveness guarantees—on faith for now, let's investigate the extent to which a sequence of random leaders is balanced. We'll keep the discussion a bit on the intuitive and informal side; it wouldn't be hard to turn this discussion into rigorous proof, but that wouldn't be the best use of our time. Let α denote the fraction of nodes that are Byzantine (in terms of our usual notation, $\alpha = \frac{f}{n}$). We saw in Section 8.4 that there's no hope of proving anything unless $\alpha < \frac{1}{2}$, so let's go ahead and assume that from here on out. The basic idea is:

1. A consequence of the law of large numbers is the hopefully intuitive fact that if you repeat an experiment with success probability p a large number of times, the long-run fraction of successes you'll see is almost always going to be roughly p.

2. Identifying a successful experiment with the selection of an honest leader, means that after enough repeated trials the long-run fraction of honest leaders is almost always going to be roughly $1 - \alpha$.

3. Because $\alpha < \frac{1}{2}$, this proportional representation of honest leaders means that the honest leaders should outnumber the Byzantine leaders in every sufficiently large window.

The parameter w in the w-balancedness condition will control how large we need to take the parameter k—the number of subsequent blocks required to regard a block as finalized— to guarantee consistency and liveness. Time-to-finalization is an important performance characteristic of a blockchain protocol, so we'd like a more quantitative understanding of just how balanced a sequence of random leaders is likely to be.

Toward a quantitative understanding. The exact guarantee on the parameter w will depend on just how far α is from $\frac{1}{2}$—the closer it is to α, the bigger the w we'll need to take to ensure sufficiently proportional representation of honest leaders.
For example, imagine that α is 49%. In a length-100 window of consecutive leaders, we would expect 51 honest nodes and 49 Byzantine nodes. There will be some variance, of course—sometimes there will be 55 honest leaders, sometimes there will be 47 (which would violate 100-balancedness), and so on. So we would not expect a random leader sequence to be 100-balanced when $\alpha = 0.49$.
In a length-1000 window, meanwhile, we would expect 510 honest nodes and 490 Byzantine nodes—now we have a margin of error of 20 between the two, and it intuitively seems that we should have a better shot at seeing a majority of honest nodes. Sometimes there might be 515 honest nodes, sometimes 505, sometimes 523, and sometimes 497 (which would violate 1000-balancedness), and so on. So we might expect a random leader sequence to be 1000- balanced with somewhat high (but not 99.9%) probability. Similarly, with length-10000 Windows, we have a buffer of 200 between the expected number of honest and Byzantine nodes to absorb the inevitable variations, and it would seem still more unlikely to have an unluckily low number (¡ 5001) of honest nodes. If this intuition is correct, given a desired failure probability δ, we should be able to take the window length large enough (as a function of α and δ) to guarantee a failure probability of at most δ.

8.6.3 Quantitative Analysis of Random Leader Selection

Exponentially small bounds. The intuition above is quite accurate. But if you really want some concrete numbers, you have to do some actual math. And if you do the math, the very cool thing that you discover is that not only is the probability of a w-balancedness violation decreasing with the window size, it's decreasing exponentially quickly. More formally:

$$\textbf{Pr}[\text{a given length-w window is at least half Byzantine}] \leqslant e^{-cw}, \quad (2)$$

where c is some constant (independent of w). (The constant c depends on α—the closer α is to $\frac{1}{2}$, the smaller the c and the slower the decrease in failure probability.) If you want to keep a concrete number in mind, think of c as being (for example) 0.1. Also, the "e" in (2) denotes the base of the natural logarithm, $2.718\cdots$.
An exponentially small failure probability like the one in (2) is good news: every time you bump up the window length w by 1, the failure probability drops by another constant factor. This is the key property behind the assertion that random leader sequences are w-balanced for reasonably small values of w (with high probability). (Remember, w controls the time to finality, and we want to take it to be as small as possible!)

Applying the Union Bound. The failure probability in (2) concerns a specific window (e.g., the leaders of rounds $101, 102, \cdots, 140$). Looking back at Definition 8.5.1, we see that it asserts something about every window of sufficiently large length (e.g., at least 40 leaders in a row), not just one window. To bridge that gap, we can use the Union Bound, which states that the probability that any bad event happens is at most the sum of their individual probabilities:

$$\textbf{Pr}[\text{at least one of the events } E_1, E_2, \cdots, E_k \text{ occur}] \leqslant \sum_{i=1}^{k} \textbf{Pr}[\text{event } E_i \text{ occurs}]$$

Or, in terms of the complementary event:

$$\textbf{Pr}[\text{none of the events } E_1, E_2, \cdots, E_k \text{ occur}] \geqslant 1 - \sum_{i=1}^{k} \textbf{Pr}[\text{event } E_i \text{ occurs}] \quad (3)$$

For us, the individual events E_i correspond to the windows of length at least w (with E_i occurring if and only if the corresponding window is at least half Byzantine leaders). The key point is that, by definition, a leader sequence is w-balanced if and only if none of these events occur.

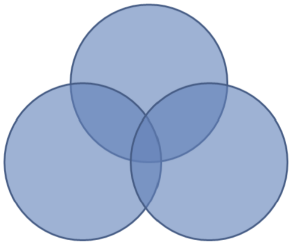


Figure 8.7: The Union Bound: the probability of the union of a bunch of events is bounded above by the sum of the probabilities of the individual events.

Now imagine that we're interested in some stretch of T consecutive rounds of the longest chain consensus (which could represent a day, a month, etc.). The number of windows (of length at least w, or otherwise) can be bounded above crudely by T^2 (with only T choices for the first and for the last round of the window). Because there are at most T^2 E_i's and (by (2)) $\textbf{Pr}[E_i] \leqslant e^{-cw}$ for every i, the Union Bound (3) tells us that

$$\textbf{Pr}[\text{a sequence of } T \text{ randomly selected leaders is } w\text{-balanced}] \geqslant 1 - T^2.e^{-cw} \quad (4)$$

Solving for w given choices of T and δ (and α). So, is the right-hand side of (4) big or small? The answer depends on our choice of w. Fix a failure probability δ that we're comfortable with, such as $\delta = 0.01$. The bound in (4)

then tells us how to set w, by setting $\delta = T^2.e^{-cw}$ and then solving for w (by taking logarithms of both sides and rearranging). When the dust settles, the punchline is:

Note:-

a sequence of T random leaders is w-balanced with probability at least $1 - \delta$, provided

$$w \geqslant c_2(\ln T + \ln \frac{1}{\delta})$$

Here, c_2 is a constant independent of w and δ. (Like c_1, it does depend on α—the closer α is to $\frac{1}{2}$, the bigger the constant factor c_2.) The exponentially small probability in (2) thus translates to a lower bound on w that involves only the logarithms of the two key parameters, the duration T of interest and the (reciprocal of the) acceptable failure probability. The most important fact about logarithms is that they grow really slowly—for example, if we're talking about the base-2 logarithm, the logarithm of 1000 is roughly 10, the log of one million is roughly 20, the log of one billion is roughly 30, and so on. This means that we can take the duration T to be quite long and the failure probability δ quite small while still getting away with palatable values of w.

Recap.

1. As we'll see in Sections 8.7–8.10, the distance k of a block from the end of the longest the chain that is required before the block should be considered finalized is controlled by the smallest w for which the generated leader sequence is w-balanced.

2. Randomly chosen leaders are w-balanced with high probability, where the parameter w depends on the fraction α of Byzantine nodes, the duration T of interest, and the desired failure probability δ (with w increasing with α, T, and $\frac{1}{\delta}$).

3. Because the lower bound on w increases only logarithmically quickly with T and $\frac{1}{\delta}$, for typical parameter values (say $\alpha = 0.33$, $T = 1000$, and $\delta = .01$) it's safe to take w (and hence k) in the double digits. (A more detailed analysis can be used to improve this conservative lower bound by a constant factor.)

8.6.4 Toward Permissionless Consensus

Longest-chain consensus is already interesting to study in the permissioned setting (scenario (PKI)), but what's really special about it is how easily it extends to the permissionless case. Looking ahead to chapters 9 and 12, let's examine exactly how the permissioned assumption was used in the analysis of this section. Rereading the informal argument in Section 8.6.2, we see that only one thing matters for the analysis: every round, the probability that a Byzantine node is selected as the leader is less than 50%. (This then leads to (almost) proportional representation of honest nodes in sufficiently long windows, all of which will then have a majority of honest nodes.)

Note:-

Key Property for Analysis of Random Leaders

There is a constant $\alpha < \frac{1}{2}$ such that, in every round, the probability that a Byzantine node is selected as the round's leader is at most α. (Also, each leader should be chosen independently of any of the previous ones.)

If this key property holds—for whatever reason, whatever the concrete implementation of longest-chain consensus—the "proportional representation" argument remains valid.21 Now, in scenario (PKI), it's easy to think of a combination of assumptions and protocol design choices that together ensure that the key property above holds: if less than half the nodes are Byzantine and each leader is selected independently and uniformly at random from the set of all nodes, then the probability that a given round has a Byzantine leader is less than 50%.

In the permissionless scenarios (PoW) and (PoS), where the set of nodes running the protocol is unknown, it's not obvious how to sample a node uniformly at random. But, if we can find a combination of assumptions and (permissionless) protocol design decisions that ensure the key property above, we'll be good to go—the generated leader sequence will be balanced with high probability, from which consistency and liveness follow.

8.7 The Common Prefix Property

With this section, we embark on the second step of the plan (Section 8.5.2), which is to show that balanced leader sequences (in the sense of Definition 8.5.1) guarantee the consistency and liveness of longest-chain consensus (provided the parameter k is set appropriately). We already know (Section 8.4.2) that balanced leader sequences are necessary for these properties (as otherwise Byzantine nodes can cancel many blocks from the end of the current longest chain); so, this part of the plan effectively shows that an unbalanced leader sequence is the only thing that could interfere with longest-chain consensus.

8.7.1 Statement of the Common Prefix Property

We'll begin with the common prefix property. Assumption (A5) immediately simplifies our lives in that, at any given time, all honest nodes are aware of exactly the same in-tree G of blocks. (Whenever an honest node becomes aware of a new block, it immediately informs everyone else, and under the assumption (A5) these messages arrive immediately.) Byzantine nodes may know about additional blocks outside of G that have been created but not yet announced.
Recall the definition of $B_k(G)$ in (1), as the longest chain of the in-tree G with the last k blocks lopped off. For this to make mathematical sense, in the presence of multiple longest chains, you should get the same thing no matter which one you choose. Equivalently, all longest chains of G must share a long common prefix, with all such chains agreeing on all but perhaps their last k blocks. The common prefix property asserts that, if the leader sequence is balanced and the parameter k is chosen appropriately, then $B_k(G)$ is guaranteed to be well-defined.

Theorem 8.7.1 Common Prefix Property of Longest-Chain Consensus

If the leader sequence $l_1, l_2, l_3, \cdots$ is ($2k+2$)-balanced and assumptions (A1), (A4'), and (A5) hold, then for every possible resulting in-tree G of blocks known to honest nodes, $B_k(G)$ is well defined.

What do we mean by "every possible resulting in-tree"? A fixed leader sequence does not uniquely pin down a corresponding in-tree, for two reasons. The first is that the Byzantine leaders can do whatever they want (e.g., choose any existing block as a predecessor or delay the announcement of a block). The second is that honest leaders are allowed to break ties however they want. The phrase "for every possible in-tree" in Theorem 8.7.1 ranges over all possible strategies by the Byzantine leaders and all possible ways of breaking ties by the honest leaders.

Theorem 8.7.1 is stated under the stronger assumption (A4') (at most one block per leader, which holds in the proof-of-work setting) rather than the weaker assumption (A4) (no limit on the number of blocks, as in the (PKI) and (PoS) scenarios). It will be clear in the proof why (A4') is sufficient for the argument but (A4) is not. There is also a second version of Theorem 8.7.1, which we won't prove, that weakens assumption (A4') to (A4) but compensates by strengthening the balancedness assumption on the leader sequence to a "balanced on steroids" assumption. The latter condition holds with a high probability for randomly chosen leader sequences provided each leader is more likely to be honest than Byzantine (as shown by a sophisticated probabilistic argument). Thus, in any of the three scenarios, with randomly chosen leaders, the common prefix property will hold with high probability (assuming k is set sufficiently large). As we'll see in Theorem 8.8.1, as long as the common prefix property holds (for whatever reason), finality follows.

8.7.2 Proof of the Common Prefix Property (with Assumption (A4'))

The proof of Theorem 8.7.1 is perhaps the most informative one of this chapter, especially for understanding the role of the assumptions we adopted in Section 8.3. The plan is to prove the (equivalent) contrapositive statement: if there's a possible outcome G for which $B_k(G)$ is not well defined, then the leader sequence can't possibly be $(2k+2)$-balanced.

The setup. So, suppose the assumptions of Theorem 8.7.1 hold and, toward a contradiction, consider the first time in which the blocks known to (all) honest nodes form an in-tree G in which $B_k(G)$ is not well defined. This means that G contains two longest chains that differ in more than their last k blocks. Let B_1 and B_2 denote the

ends of these two chains and B^* the least common ancestor of B_1 and B_2 (if nothing else, the genesis block B_{gen}). See Figure 8.8. The plan is to show that the leader sequence could not have been $(2k+2)$-balanced, which would be a contradiction.

Life would be simple if B^* were produced by an honest leader (and therefore announced immediately). The proof will be a little more complicated than you might have expected to account for the possibility that B^* was produced by a Byzantine leader and possibly announced well after the round in which it was created.23 To deal with this, trace back from B^* toward the genesis block to the most recent block B_0 that was created by an honest leader. (If B^* happened to be produced by an honest leader, then $B_0 = B^*$.) The block B_0 must exist because, if nothing else—by assumption (A1)!—the genesis block B_gen was honestly created. Because B_0 is the most recent ancestor of B^* produced by an honest leader, all of the blocks after B_0 up to and including B^* must have been produced by Byzantine leaders.

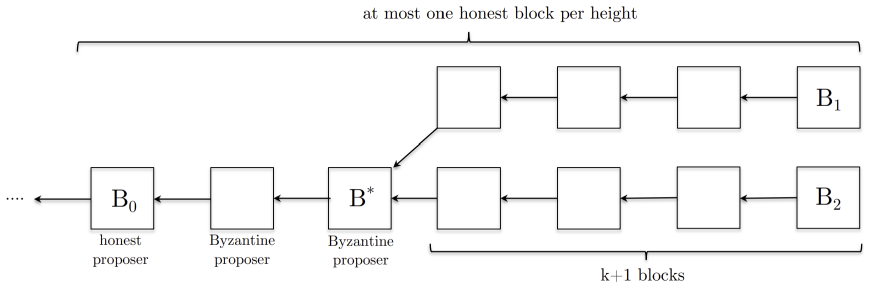


Figure 8.8: Proof of Theorem 8.7.1: if there are two longest chains that disagree in more than their last k blocks, the window of $\geqslant 2k+2$ leaders following the production of B_0 must have been at least half Byzantine.

Let's pause for a quick proposition. Recall that the height of a block is the number of hops required to get back to the genesis block (equivalently, the length of the chain from the genesis block to that block).

Proposition 8.7.1 Honest Blocks Having Increasing Heights

Under assumption (A5), the heights of honestly produced blocks are strictly increasing over time.

Proof: Suppose that the blocks B_i and B_j are produced by honest leaders i and j. There's only one leader per round, so one of i or j is chosen as a leader first—let's say i, with its block B_i at height h. Because i is honest, it announces its block B_i immediately, and by assumption (A5), node j finds out about it immediately. At the subsequent round in which j creates B_j , because j is honest, B_j extends one of the longest chains that j is aware of at that time. Because j is aware of a block at height h (namely, B_i) at that time, its block B_j will have a height of at least $h+1$. 😊

In particular, there cannot be two honestly produced blocks at the same height:

Corollary 8.7.1 At Most One Honest Block Per Height

If assumption (A5) holds and blocks $\hat{B}$ and $\tilde{B}$ were both produced by honest leaders, then $\hat{B}$ and $\tilde{B}$ have different heights.

Returning to the scenario depicted in Figure 8.8, let r_0 denote the round (with an honest leader) in which the block B_0 was created (and announced), and consider the sequence of leaders selected in rounds after r_0. The plan is to prove an upper bound on the number of honest leaders and an equal lower bound on the number of Byzantine leaders following round r_0, which together show that there is a long window comprising at least 50% Byzantine leaders (a contradiction). So, let h_0, h^*, and h denote the heights of B_0, B^*, and the ends of the longest chains (B_1 and B_2), respectively. We can finish the proof of Theorem 8.7.1 by showing that: (i) there have been at least $2k+2$ leaders selected after round r_0; (ii) at most h-h_0 leaders selected after round r_0 could have been honest; and (iii) at least $h-h_0$ leaders selected after round r_0 must have been Byzantine. All three of these facts follow from some short observations:

1. Because B_0 was only created in round r_0, every block with B_0 as an ancestor was created in a round subsequent to r_0.

2. Because there are at least $2k+2$ blocks with B_0 as an ancestor (at least $k+1$ blocks each on the chains ending at B_1 and B_2), and because (by assumption (A4')) each of these blocks was produced by a distinct leader, the window of leaders following round r_0 has length at least $2k+2$.

3. By Proposition 8.7.1 and Corollary 8.7.1, every honest leader of a round subsequent to r_0 created (and announced) a block at a height greater than h_0, with at most one honestly produced block per height.

4. Because there have been no honestly produced blocks at heights higher than h, the total number of honest leaders subsequent to round r_0 thus far is at most $h - h_0$.

5. By the definition of B_0, the $h^* - h_0$ blocks between B_0 and B^* (excluding B_0 but including B^*) were all produced by Byzantine leaders.

6. At each height $h^*+1, h^*+2, \cdots, h$, at least two blocks have been produced (one on the chain from B^* to B_1, a second one on the chain from B^* to B_2).

7. Because there is at most one honestly produced block per height, at each height $h^*+1, h^*+2, \cdots, h$, there is at least one block produced by a Byzantine leader. Thus, the total number of blocks produced by Byzantine leaders after round r_0 is at least $(h^* - h_0) + (h - h^*) = h - h_0$.

8. By assumption (A4'), at least $h - h_0$ distinct Byzantine leaders (after round r_0) were required to produce these blocks.

9. Because there are at most $h - h_0$ honest leaders and at least $h - h_0$ Byzantine leaders after round r_0, the window of leaders from round $r_0 + 1$ to the current round is at least half Byzantine.

10. Thus, the leader sequence is not $(2k+2)$-balanced. This completes the proof of Theorem 8.7.1.

8.8 Finality of Longest-Chain Consensus

We introduced finality back in Section 8.3.4, as the consistency of an honest node with its future self. (As opposed to consistency between different honest nodes at a given moment in time, which is currently trivialized by our assumption (A5) but is also the easier consistency property to establish for longest-chain consensus.) In our discussion of BFT-type consensus protocols (chapters 2–7), the finality property was implicit in our description of the SMR problem, in particular the "append-only" requirement for honest nodes' local histories. As we've seen, in longest-chain consensus, there is a very real risk of blocks getting kicked off of the longest chain and it is not at all obvious whether finality should hold with any value of the parameter k.
Happily, the proof of the common prefix property (Theorem 8.7.1) has already done all the heavy lifting required to establish the finality of the longest-chain consensus. In fact, finality reduces to the common prefix property—if the latter holds (for whatever reason), the former automatically does as well.

Theorem 8.8.1 Finality of Longest-Chain Consensus

Let $G_1 \subseteq G_2 \subseteq \cdots \subseteq G_T$ denote a sequence of in-trees, with each in-tree G_i having one more block than the previous one G_{i-1}. If the common prefix property (with a given value of k) holds in each of $G_1, G_2, \cdots, G_T$, then:

$$B_k(G_1) \subseteq B_k(G_2) \subseteq \cdots \subseteq B_k(G_T). \quad (5)$$

You should think of the sequence $G_1 \subseteq G_2 \subseteq \cdots \subseteq G_n$as the in-tree of blocks known to (all) honest nodes at each of the rounds $1, 2, \cdots, T$. (By assumption (A5), all honest nodes know about exactly the same blocks. Honest nodes never forget about blocks, so each in-tree contains the previous one. If multiple blocks are announced in the same round, imagine that they are added to the in-tree one by one in an arbitrary order.) Then, the expression (5) asserts finality: once a block is on the longest chain with at least k blocks following it (at hence considered confirmed), it will continue to be so forevermore. Said differently, blocks only get added to $B_k(G)$ as the current in-tree G grows, never removed. The statement of Theorem 8.8.1 does not refer to any of

our assumptions (A1)–(A5), or to any assumptions about the leader sequence. None are necessary—the proof is a simple combinatorial argument about in-trees, and it shows that as long as the common prefix property holds throughout the sequence (for whatever reason), and finality also holds. For example, the conclusion of Theorem 8.8.1 will hold under the same assumptions that are stated in Theorem 8.7.1.

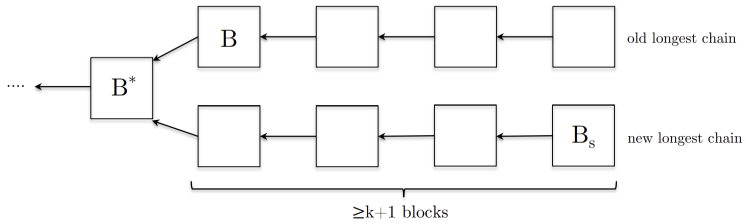


Figure 8.9: Proof of Theorem 8.8.1: the only way that a block that is at least k blocks deep on the longest chain can ever get rolled back is through a failure of the common prefix property.

Proof of Theorem 8.8.1: We proceed by contradiction. Suppose that the common prefix property (with parameter k) holds in each of $G_1, G_2, \cdots, G_T$ (and hence $B_k(G_i)$ is well defined for all i), but that finality (5) fails. This means that there's a block B that is confirmed at some step i (i.e., $B \in B_k(G_i)$) but gets rolled back at some later step $j > i$ (i.e., $B \notin B_k(G_j)$). Because $B \in B_k(G_i)$, the block B belongs to (the common prefix of) every longest chain of the in-tree G_i. Because $B \in B_k(G_j)$, there is at least one longest chain that excludes block B. Let $s \in \{i+1, i+2, \cdots, j\}$ denote the smallest index such that B is excluded from at least one longest chain of G_s. The new block B_s added to G_{s-1} to form G_s cannot have extended an existing longest chain (because that chain would then, like its prefix, include B), so it must have created a new longest chain, tied in length with the longest chains of G_{s-1} but also excluding the block B (see Figure 8.9, which may look hauntingly familiar). Pick an arbitrary longest chain C_1 of G_{s-1} and let C_2 denote the chain in G_s that ends with the new block B_s. Both C_1 and C_2 are the longest chains of G_s. Because B is more than k blocks deep on the incumbent longest chain C_1 and missing from the new longest chain C_2, the least common ancestor B of C_1 and C_2 predates B and hence is more than $k+1$ blocks deep on C_1 and C_2. This means that C_1 and C_2 differ in at least their last $k+1$ blocks. Because C_1 and C_2 are both the longest chains of G_s, G_s must then fail to satisfy the common prefix property, contradicting our initial assumption. This contradiction completes the proof that the common prefix property implies finality.

We noticed already in Section 8.4.2 that embracing forks and resolving them in-protocol would seem to threaten finality. So it's pretty cool and somewhat surprising that finality holds for longest-chain consensus under reasonably palatable assumptions (and an appropriately large value of k)!

8.9 Liveness of Longest-Chain Consensus

No analysis of a consensus protocol is complete without also addressing liveness, and proving that (under our standing assumptions) progress is guaranteed. We'll be using the same definition of liveness that we used in the last chapter to analyze Tendermint.

Note:-

Liveness (Weak Version)

If a transaction is known to all honest nodes, that transaction is eventually added to every honest node's local history.

We'll establish liveness under the same balancedness condition used in our proof of the common prefix property (Theorem 8.7.1). For the liveness analysis, we won't need to worry about the distinction between assumption (A4) and its stronger version (A4')—the argument will work assuming only (A4) and that the common prefix property holds (for whatever reason).

Theorem 8.9.1 Liveness of Longest-Chain Consensus

If assumptions (A4) and (A5) hold and the leader sequence $l_1, l_2, l_3, \cdots$ is sufficiently long and $(2k+2)$-balanced, and if the common prefix property holds in the corresponding sequence of in-trees, every transaction that is at some point known to all honest nodes will eventually be included in $B_k(G)$, where G denotes the in-tree of blocks known to (all) honest nodes.

Because the common prefix property implies finality (see Theorem 8.8.1), the transaction in question will belong to the set $B_k(G)$ of finalized blocks forevermore.

Proof of Theorem 8.9.1: The plan is to prove that honestly produced blocks get finalized (i.e., added to $B_k(G)$) infinitely often. This implies that, if a transaction tx is known to all honest nodes at some round r, there will be a future moment in time at which a block created by an honest leader in a round after r gets added to $B_k(G)$. Because an honest leader includes all the outstanding transactions that it knows about in its block, this block will contain tx (unless tx already belongs to an earlier block of $B_k(G)$).
Next, let's invoke the balancedness assumption on the leader sequence. Split the sequence into groups of $2k+2$ consecutive leaders each, and call each group an epoch. Because the leader sequence is $(2k+2)$-balanced, every epoch contains more honest leaders than Byzantine leaders—at worst, $k+2$ honest leaders and k Byzantine leaders.
Recall that the height of a block is the length of the chain from genesis to that block. The length of the longest chain in an in-tree is therefore the same as the maximum block height. Because we're assuming that (A5) holds, we can invoke Proposition 8.7.2 to conclude that, in every epoch, the length of the longest chain grows by at least $k+2$ (i.e., by at least one for each honest node in the epoch).
Fast forward to the end of the Tth epoch (for a parameter T), and consider a longest chain at that time, which must contain at least $(k+2)T$ blocks. By assumption (A4), each Byzantine leader from the first T epochs contributes at most one block to this chain. Because at most kT leaders from the first T epochs were Byzantine, this longest chain contains at least $(k+2)T - kT = 2T$ blocks produced by honest leaders. Because the common prefix property holds (by assumption) for the current in-tree GT of blocks known to all honest nodes, $B_k(G_T)$ is well defined and contains at least $2T-k$ of this $2T$ honestly produced blocks. Thus, as the parameter T increases (and remembering that k is a fixed constant like 100, independent of T), honestly produced blocks must be added to $B_k(G)$ infinitely often. This fulfills our proof plan and completes the liveness analysis of longest-chain consensus.

8.10 Chain Quality of Longest-Chain Consensus

8.10.1 Stronger Liveness Guarantees

Theorem 8.9.1 establishes the basic liveness guarantee of longest-chain consensus—under our standing assumptions, transactions known to all honest nodes will eventually be finalized. What else could we want? There are a few different ways we could try to strengthen this liveness guarantee.

1. Replace the assumption that all honest nodes know about a transaction with the weaker assumption from Chapter 2 that at least one honest node knows about it. As noted in the previous section, this stronger liveness guarantee generally doesn't hold for longest-chain consensus under our standing assumptions (exercise).

2. Strengthen the "will eventually be finalized" guarantee to a quantitative guarantee that gives concrete bounds on latency (i.e., on time-to-finalization) as a function of the relevant parameters (e.g., how far the fraction α of Byzantine nodes/hashrate/stake is below 50%). This is a quite an interesting topic, but exploring it would take us too far afield; see e.g. for entry points to the academic literature on such latency guarantees.

3. Strengthen the "honest blocks get added to $B_k(G)$ infinitely often" guarantee to a concrete lower bound on the fraction of blocks of $B_k(G)$ that was produced by honest nodes (again, as a function of the relevant parameters). This fraction is known as the chain quality, and it is the strengthening that we will discuss here.

If the fraction of honest nodes/hashrate/stake is barely above 50%, the chain quality of longest-chain consensus isn't really any better than what Theorem 8.9.1 suggests (namely, that the chain quality is more than 0%). But does it help if, say, 60% of the nodes are honest?

8.10.2 Generalizing the Proportional Representation Argument

Let's return to the case of randomly chosen leaders, as in Section 8.6. Recall the first two parts of the intuition for the analysis of random leaders in Section 8.6.2 (and the corresponding proofs in Section 8.6.3):

1.

1. A consequence of the law of large numbers is the hopefully intuitive fact that if you repeat an experiment with success probability p a large number of times, the long-run fraction of successes you'll see is almost always going to be roughly p. 2. Identifying a successful experiment with the selection of an honest leader, means that after enough repeated trials the long-run fraction of honest leaders is almost always going to be roughly $1-\alpha$, where $\alpha = \frac{f}{n}$ denotes the fraction of nodes that are Byzantine. In our basic analysis in Section 8.6, we then used the fact that $\alpha < \frac{1}{2}$ (and hence $\alpha < 1-\alpha$) to conclude that honest leaders should outnumber the Byzantine leaders in every sufficiently large window. More generally, the exact same argument shows that the fraction of honest leaders should be very close to $1-\alpha$ in every sufficiently large window. To make this precise:

Definition 8.10.1: (w,ρ)-Balanced Leader Sequence

A sequence $l_1, l_2, l_3, \cdots \in \{H, A\}$ is $(w,)$-balanced if, in every window $l_i, l_{i+1}, \cdots, l_{j-1}, l_j$ of length at least w, the number of A's is less than $\rho.(j-i+1)$.

The basic w-balanced condition (Definition 8.5.1) is the special case of the (w,ρ)-balanced condition with $\rho = \frac{1}{2}$.
The hope is that if, for example, only 40% of the nodes are Byzantine, then in every sufficiently long window less than 41% of the chosen leaders are Byzantine (with high probability). And indeed, the "proportional representation" analysis from Section 8.6.3, repeated verbatim, culminates in the following analog of (4):

$$\textbf{Pr}[\text{a sequence of } T \text{ randomly selected leaders is } (w, \alpha+\varepsilon)\text{-balanced}] \geqslant 1 - T^2.e^{-cw} \quad (6)$$

(The "e^{-cw}" failure probability term comes from the law of large numbers argument summarized in (2) and applied to a single length-$\geqslant w$ window, and the "T^2" term comes from a Union Bound over all possible length-$\geqslant w$ windows.)

Glossary of notation. In (6), w is the minimum window length we're interested in (perhaps 50 or 100), T is the length of the entire leader sequence that we're looking at (e.g., the leaders chosen over the course of a day or a week), α is the fraction of nodes that are Byzantine (e.g., 40%), ε is a parameter of our choosing that is close to 0 (e.g., 0.1 or 0.01), and c is a constant independent of T, w, and α (like .1). The constant c does depend on our choice of ε, with smaller values of ε (i.e., more stringent proportional representation demands) leading to smaller values of c (i.e., a less rapid decay in the failure probability as a function of the minimum window length w). You can think of the basic analysis from Section 8.6.3 as the special case in which $\varepsilon = \frac{1}{2} - \alpha$. As in Section 8.6.3, we then have:

Note:-

a sequence of T random leaders is $(w, \alpha+\varepsilon)$-balanced with probability at least 1−, provided

$$w \geqslant c_2(\ln T + \ln \frac{1}{\delta})$$

Again, the constant c_2 depends on ε (with c_2 increasing as ε gets closer to 0) but is independent of all other parameters. As expected, larger minimum window lengths are required to achieve smaller failure probabilities (δ) and smaller tolerances around deviations from proportional representation (ε).

8.10.3 From Liveness to Chain Quality

Now that we have our generalized balancedness condition (Definition 8.10.1), we can simply repeat our liveness analysis (Theorem 8.9.1) to obtain a stronger chain quality guarantee.

Theorem 8.10.1 Chain Quality of Longest-Chain Consensus

If assumptions (A4) and (A5) hold and the leader sequence $l_1, l_2, l_3, \cdots$ is sufficiently long and $(2k+2, \alpha+\varepsilon)$-balanced, then the fraction of blocks on a longest chain that was produced by honest leaders is always as least

$$\frac{1-2\alpha-2\varepsilon}{1-\alpha-\varepsilon} \quad (7)$$

You can think of Theorem 8.9.1 as the special case of Theorem 8.10.1 in which $\alpha < \frac{1}{2}$ and ε is chosen infinitesimally smaller than $\frac{1}{2} - \varepsilon$ (so that $\alpha + \varepsilon < \frac{1}{2}$ and the fraction in (7) is just barely positive). For example, if 60% of the nodes are honest (and ε is small), Theorem 8.10.1 asserts that at least roughly one-third of the blocks in $B_k(G)$ were produced by honest leaders. If two-thirds of the nodes are honest, the chain quality improves to roughly 50%.

Proof of Theorem 8.10.1: As in the proof of Theorem 8.9.1, break the leader sequence into epochs that each consist of $2k+2$ consecutive leaders. By the $(2k+2, \alpha+\varepsilon)$-balancedness assumption and Proposition 8.2, the length of the longest chain grows by at least $(2k+2)(1-\alpha-\varepsilon)$ with every epoch (i.e., by at least one for each honest leader of the epoch). After T epochs, there are at least $(1-\alpha-\varepsilon)(2k+2)T$ blocks on a longest chain, and (by assumption (A4)) at most $(\alpha+\varepsilon)(2k+2)T$ of these could have been contributed by Byzantine leaders (i.e., at most one block per such leader). This means that the fraction of blocks on this chain that were honestly produced are at least

$$\frac{(1-\alpha-\varepsilon)(2k+2)T - (\alpha+\varepsilon)(2k+2)T}{(1-\alpha-\varepsilon)(2k+2)T} = \frac{1-2\alpha-2\varepsilon}{1-\alpha-\varepsilon}$$

as promised.

8.10.4 Chain Quality and Selfish Mining

How should you feel about the chain quality guarantee in (7)? The good news is that as α tends to 0, the chain quality tends to 100%. On the other hand, it's a bit disappointing that with 67% honest nodes Theorem 8.10.1 promises only a chain guarantee of 50%, rather than 67%. Shouldn't honest nodes get their fair share of blocks on the longest chain? Maybe we can be smarter and rework the proof of Theorem 8.10.1 so that its chain quality guarantee of (roughly) $\frac{1-2\alpha}{1-\alpha}$ improves to $1-\alpha$?

Alas, with arbitrary Byzantine strategies and arbitrary tie-breaking by honest nodes, the analysis of the chain quality of longest-chain consensus in Theorem 8.10.1 is tight—for any $\alpha < \frac{1}{2}$, the chain quality might be as low as $\frac{1-2\alpha}{1-\alpha}$. We'll see proof of this fact in chapter 10 in the context of "selfish mining," which is a specific strategy by Byzantine nodes (involving the delayed announcements of blocks) designed to kick as many honestly produced blocks off the longest chain as possible.

It is possible to add a bit more complexity to longest-chain consensus so that the chain quality is guaranteed to be roughly $1-\alpha$ (in addition to consistency and liveness in the synchronous model whenever $\alpha < \frac{1}{2}$).

8.11 Longest-Chain Consensus in the Partially Synchronous Model

8.11.1 Partial Synchrony

This entire chapter we've been operating under assumption (A5), meaning that all communication between honest nodes happens instantly (equivalently, the synchronous model with delay parameter $\Delta = 0$). In Chapter 9 we'll see why all of Theorems 8.7.1, 8.8.1, 8.9.1, and 8.10.1 extend to the synchronous model with minimal loss, provided the maximum network delay Δ is much smaller than the typical duration of a round. That's great, but at the same time, Tendermint (chapter 7) spoiled us by offering consistency (always) and liveness (eventually) in the more realistic partially synchronous setting, meaning even in the presence of denial-of-service attacks and network partitions. What guarantees are offered by longest-chain consensus here? More formally, recall (from Chapter 6) that the partially synchronous model assumes a global shared clock and involves two parameters, an unknown global stabilization time (GST) and a known bound Δ on the maximum message delay after GST has passed. (Intuitively, the network begins under attack, the attack eventually ends, and after that, the network resumes normal operation.) Before GST has passed—and nobody knows whether it has passed or not, only that

it will happen eventually—messages can be delayed arbitrarily.

Figure 8.10: When there are no message delays or Byzantine nodes, there are no forks.

8.11.2 A Counterexample to Finality

Longest-chain consensus breaks down badly in the partially synchronous model, unfortunately, even when there are no Byzantine nodes at all. Take finality (Theorem 9.1), for instance, and set the security parameter k as big as you like. Finality asserts that once a block is at least k blocks deep on a longest chain, it will always be at least k blocks deep on a longest chain.

Now imagine that longest-chain consensus (in the permissioned and PKI setting, with no Byzantine nodes) is happily chugging along, with no forks whatsoever, producing a chain of length l (Figure 8.10) that is known to all the (honest) nodes. Suppose that we're still pre-GST, though, and immediately after the creation of the last block B_l there is a network partition. In a network partition (Figure 8.11)—which you might remember from our discussion of the CAP Principle in Chapter 6—the nodes are split into two groups (A and B), with all intragroup messages delivered immediately and messages between groups delayed indefinitely (until after GST).

Intuitively, in longest-chain consensus, the two sides of the network partition are going to continue operating independently, alone in their own universes. For example, suppose the next round is r, and the leader for round r is a node i in group A. In the permissioned version of longest-chain consensus with the PKI assumption, everyone (both in group A and in group B) knows that i is the round-r leader. Node i is honest, so it dutifully produces a block B_{l+1} that extends the longest chain that it knows about and immediately announces that block to all the other nodes. Nodes in A hear the announcement and from their perspective, the protocol has continued to hum along normally. Nodes in B hear nothing. From their perspective, something has gone wrong, but it's not clear what—for example, for all they know, node i is Byzantine and never sent any block announcements to any nodes of B.

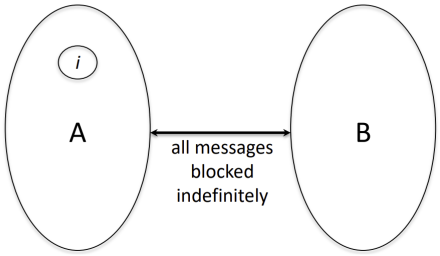


Figure 8.11: During a network partition (e.g., due to a denial-of-service (DoS) attack), two disjoint sets of nodes (A and B) are unable to communicate with each other.

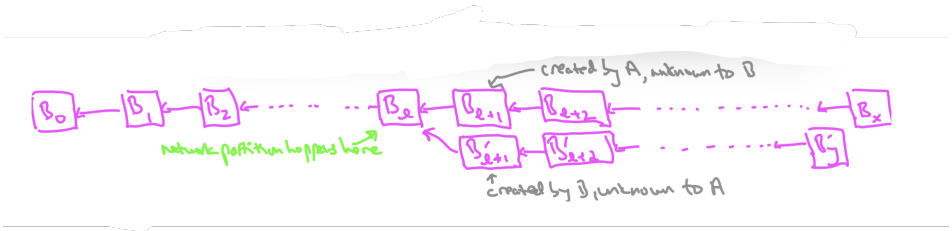


Figure 8.12: With a network partition, the two sides of the partition independently create two competing chains.

At this point, nodes in group A view block B_{l+1} as the end of the longest chain, while nodes in group B don't know about B_{l+1} and think that B_l is still the end of the longest chain. Now suppose that the leader of round $r+1$ is a node from group B. That (honest) node will dutifully extend the longest chain that it knows about, which means that it will produce a second block B'_{l+1} that extends B_l. Now, even though there are no Byzantine nodes, the blockchain has suffered a fork. The leader of round $r+1$ will broadcast its block B'_{l+1} to everybody, but only the nodes in group B will hear about it. As long as the network partition persists, the pattern recurs. In every round with a leader from group A, a new block is added to the end of the top branch (the only branch that group-A nodes know about), and similarly for leaders from group B and the bottom branch. When the network partition eventually ends and all of the stale intergroup block announcements are delivered, all nodes will realize what the full blockchain looks like (as in Figure 8.12). (Up to this point, every node thought that the blockchain consisted of only a single chain.) The nodes will realize that they've been doing redundant uncoordinated work and, worse, that the shorter of the two branches will have to be discarded. For example, imagine that the security parameter k is 50 and that the network partition goes on so long that the nodes of groups A and B add 125 and 119 blocks to their respective branches. (Remember GST is unknown and arbitrarily large. Thus no matter how big k is chosen, a network partition can go on long enough for the creation of two competing branches that are both longer than k.) The nodes in group A will breathe a sigh of relief, discard the (shorter) competing branch, and continue with the same view of the longest chain as they had before. The nodes in B, however, previously regarded the last $119-k=69$ blocks of their branch as finalized (i.e., belonging to what they thought $B_k(G)$ was), but now all of those blocks have to be rolled back. This is a violation of finality.

8.11.3 Discussion

The fact that longest-chain consensus fails to guarantee consistency in the partially synchronous model (in contrast to BFT-type protocols like Tendermint) is arguably its most serious flaw. Longest-chain consensus is still useful, of course—it powers many of the world's biggest blockchains—but it's important that everyone understands its weaknesses. The breakdown of consistency in longest-chain consensus in the partially synchronous model ties back to our discussion of fundamental consistency-liveness tradeoffs (see the FLP impossibility result in chapters 4 5) and the different failure modes of different types of blockchain consensus protocols. We know (from FLP) that we can't guarantee both consistency and liveness during an attack. The best we can hope for is to maintain one of them while under attack and to recover the other quickly after the attack ends (i.e., post GST). BFT-type protocols like Tendermint guarantee consistency while under attack, so it's no surprise that they temporarily give up on liveness (and, for this reason, generally fail in practice by stalling for an extended period of time).

Meanwhile, longest-chain consensus still has a form of liveness during a network attack.34 In our example above, 75 new blocks are finalized (and not later rolled back) during the network partition! A BFT-type protocol, meanwhile, would finalize zero new blocks during this time. The resiliency of liveness in longest-chain consensus during a network attack would then suggest that consistency must be given up instead, as we verified with the concrete counterexample above. For this reason, longest-chain blockchain protocols tend to fail not by stalling but through large-scale chain reorganizations and the consequent reversal of once-thought-confirmed transactions.

Longest-chain consensus makes very different trade-offs than BFT-type protocols and, as such, is an innovation even from the perspective of traditional (permissioned + PKI) consensus protocols. The traditional 20th-century literature on consensus protocols focused almost entirely on protocols that favored safety over liveness (like BFT-type protocols). It's easy to see why—consistency, as a safety property (i.e., bad things never happen), was historically viewed as mission-critical and non-negotiable. Because liveness guarantees are inherently about good things happening eventually, it was natural to allow for the "eventually" to encompass "post-GST/attack."

Longest-chain consensus shows that the classical literature was missing some interesting and non-trivial points in the consensus protocol design space, and that favoring safety over liveness is not the only option. If the application demands it, you can use a consensus protocol (like longest-chain consensus) that continues to make progress when under attack, with the understanding that some honest nodes may have to roll back some of the progress that they thought they had made.

8.12 Toward Permissionless Consensus

Permissionless longest-chain consensus. This chapter has focused on longest-chain consensus in the safe confines of the permissioned setting with the PKI assumption, for the following reasons: (i) continuity with chapters 2–7; (ii) to focus on the basic consistency and liveness guarantees of longest-chain consensus without worrying about the additional challenges that arise in the permissionless setting (primarily "sybils"); (iii) to appreciate that

longest-chain consensus is an interesting part of the design space even in the classical permissioned + PKI setting.

All that said, the real claim to fame for longest-chain consensus is its extensions to the permissionless setting, in which the protocol has no idea which nodes might be running it. This is obviously a much different scenario than traditional applications like database replication, in which all the nodes would typically be servers bought in advance and operated by a single entity. To viscerally appreciate the difference, I encourage you to spin up a full node for a blockchain protocol like Bitcoin or Ethereum—no one can stop you, you can literally just download the appropriate software and do it, right now.

Bitcoin's fundamental breakthrough was a solution to permissionless consensus. This involved both the invention of a new approach to consensus (longest-chain consensus) and a novel method for "sybil-resistance" (called "proof-of-work") that extends the guarantees of longest-chain consensus from the permissioned to the permissionless setting.35 We now understand that there are also other viable approaches to permissionless consensus, including ones that use BFT-type consensus protocols and alternative approaches to sybil-resistance (like "proof-of-stake"). We'll get into much more detail on all of this in chapters 9 (for proof-of-work) and 12 (for proof-of-stake).

The essence of the analysis. So what prevents the description and analysis of longestchain consensus in this chapter from applying immediately to the permissionless setting? Where in this chapter did we lean on the permissioned assumption?

The answer lies in the leader selection step of longest-chain consensus (step (2a) in the description in Section 8.2.2). The abstract description is silent on how this mapping (from rounds to leaders) is made, in effect assuming that it's carried out by some black box (Figure 8.13). What properties of this black box were necessary for the proof of the basic consistency and liveness properties in Sections 8.7–8.10? Three things:

Note:-

Required Properties of the Leader Selection Box

1. (Same as assumption (A2)) It is easy for all nodes to verify whether a given node is the leader of a given round.
2. (Same as assumption (A3)) No node can influence the probability with which is selected as the leader of a round.
3. (Required hypothesis for Theorems 8.7.1, 8.9.1, and 8.10.1) The sequence of leaders are sufficiently balanced.

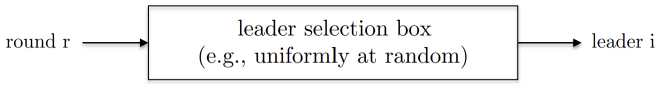


Figure 8.13: Step (2a) of longest-chain consensus is effectively a black box that maps round numbers to leader nodes.

For example, the analysis in Sections 8.7–8.10 makes no reference to the number of nodes n—all the proofs work as long as the sequence of As and Hs generated by the leader selection box is sufficiently balanced.
Depending on the context, "sufficiently balanced" could mean the w-balanced condition in Definition 8.5.1 (for Theorems 8.7.1 and 8.9.1), the version of that definition parameterized by α and ε (Definition 8.10.1, in service of Theorem 8.10.1). This is a zoo of different conditions, but there's a common sufficient condition that implies all of them (with high probability):

Note:-

Key Sufficient Condition for Consistency and Liveness of Longest-Chain Analysis

Leaders in different rounds are chosen independently and, in each round, the probability that the leader is Byzantine is at most $\alpha < \frac{1}{2}$

In other words, if you push the button on the black box to generate a new leader, the leader that pops out is more likely, to be honest than Byzantine (with each selection independent of previous ones). This condition is the only thing we ever used in the "proportional representation" arguments in Section 8.6.3 (as you should check), and more generally it implies all the balancedness conditions above (with high probability, and assuming that parameters like w are chosen appropriately).

All the properties of longest-chain consensus that we care about (Theorems 8.7.1, 8.8.1, 8.9.1, and 8.10.1) thus boil down to having a sufficiently balanced leader sequence, and this, in turn, boils down to making sure that the key sufficient condition above holds. So how do we do that?

A permissionless leader selection box? In the permissioned setting with n nodes and the PKI assumption, the answer is straightforward: assume that the fraction $\alpha = \frac{f}{n}$ of nodes running the protocol that are Byzantine is less than $\frac{1}{2}$, and in each round select one node as the leader uniformly at random (with probability $\frac{1}{n}$ for each node), independently of the other rounds.

Selecting a node uniformly at random would seem to be a fundamentally permissioned idea. In the permissionless setting, where you have no idea which nodes are running the protocol (e.g., the protocol doesn't know n), it's not clear how to select a node uniformly at random. A permissionless version of the leader selection black box, satisfying the key sufficient condition above (and also assumptions (A2) and (A3)), is thus the missing ingredient to a permissionless version of longest-chain consensus that satisfies all of the guarantees that we proved in this chapter. Chapter 9 presents one such permissionless black box (under the assumption that Byzantine nodes always contribute less than half of the overall computational power devoted to the protocol) and Chapter 12 another (under the assumption that the blockchain has a native cryptocurrency and that Byzantine nodes always contribute less than half of the overall amount of currency that has been staked in an appropriate smart contract). Once these permissionless leader selection boxes are in place, the consistency and liveness guarantees from this chapter carry over immediately.

Hopefully, you feel like we're close to achieving permissionless consensus with provable consistency and liveness. And we are! Next chapter supplies (one option for) the missing ingredient, sybil-resistance through proof-of-work.

Chapter 9

Permissionless Consensus and Proof-of-Work

9.1 Permissionless Consensus

Recap of the permissioned setting. Thus far we've been studying consensus protocols in the safe confines of the permissioned setting, in which the nodes running the protocol are fixed and known in advance. This scenario made perfect sense when computer science researchers started developing a theory of consensus protocols in the 1980s when a typical application might be something like database replication (with a single company buying a bunch of nodes to each store a copy of the database in the interests of very high uptime). Thus far, we've also been making a PKI (public key infrastructure) assumption. That is, we're assuming that secure signature schemes exist, that each node has its public-private key pair, and that all nodes' public keys are known to all nodes at the beginning of the protocol (with each node signing its messages and verifying signatures on all received messages). This is a "trusted setup" assumption, meaning we just assume that all nodes' public keys were somehow shared correctly in advance of the protocol's commencement (just as we assumed that all nodes somehow obtained a correct version of the protocol's code). This trusted setup assumption is also totally reasonable in our running example of database replication by a centralized organization. Tendermint (Chapter 7) is a canonical example of a permissioned consensus protocol.

Open participation and permissionless protocols. The most famous blockchain protocols, such as Bitcoin and Ethereum, do not operate in the permissioned setting. To viscerally appreciate this point, I encourage you to take a break from this chapter and spin up on your laptop or desktop a full node that validates the Bitcoin or Ethereum blockchains. These blockchains have been running for years and have no idea who you are, and yet you can simply download software, sync your machine with the blockchain so far, and join the party. This is obviously very different from the classical permissioned setting. Remember in Chapter 1 when I described a mental model of a blockchain as a (virtual) computer in the sky, with no owner or operator? Really what this meant was that this computer has lots of operators (coordinated via a consensus protocol), and moreover, you yourself can become one of these operators, if you wish. This is the vision of permissionless consensus, and it is perhaps the most radical and audacious conceit in the original Bitcoin protocol.

The challenges of permissionless consensus. A permissionless consensus protocol has an unknown and possibly ever-changing set of nodes running it, which would seem to throw an immediate wrench into, for example, a Tendermint-style protocol. In Tendermint with 100 nodes, nodes will wait to collect and see evidence of 67 votes before proceeding with any action. If you don't know how many nodes are there, how do you know how many votes to collect? Similarly, without knowing which nodes are running the protocol, how would you ever implement the round-robin approach to leader selection (in Tendermint, or in longest-chain consensus)? It also makes sense to think about permissionless consensus protocols as operating via broadcast channels—perhaps implemented via a gossip protocol in a free-to-join peer-to-peer network—rather than point-to-point communication. So, we'll imagine that whenever an honest node sends a message, it automatically broadcasts the messages to all other nodes. (We don't want to impose any unjustified restrictions on what Byzantine nodes can do, so we'll continue to allow them to, for example, send conflicting messages to different sets of honest nodes.) Similarly, clients

(in the sense of a state machine replication protocol) are assumed to submit their transactions via a broadcast channel that is listened in on by all the nodes that are currently running the protocol. This "broadcast-only" communication restriction shouldn't bother you too much—if you go back over our pseudocode for the Tendermint protocol in chapter 7, for example, you'll see that the nodes are basically already communicating via broadcast messages, anyway (e.g., announcing or echoing quorum certificates).

The dream of permissionless consensus is an ambitious one, and in light of the seeming incompatibilities between the constraints of the permissionless setting and the consensus protocols that we've seen thus far, you'd be right to question permissionless consensus protocols with provable guarantees could even exist!

Such protocols do exist, if they didn't, there'd be no chapter series! but, we'll need some additional ideas. As mentioned, an immediate obstacle to implementing a BFT-type protocol in a permissionless setting is the seeming impossibility of supermajority voting when the set of voters is unknown. A second issue, relevant to both BFT-type protocols and longest-chain consensus, is how to select leaders (i.e., which node gets to propose the next block). "Round-robin order" doesn't appear to make sense with an unknown and ever-changing set of nodes. "Uniformly random selection" also doesn't seem to make sense—if there were 100 nodes, in each round a given node should be selected as a leader with 1% probability. If there were 1000 nodes, it should be a 0.1% probability. But if the protocol doesn't know how many nodes there are, how can it know the appropriate probability with which to select a node?

9.2 Random Sampling and Sybil Attacks

9.2.1 Why Permissionless Random Sampling Is Useful

The key idea for transforming both BFT-type and longest-chain consensus protocols into permissionless protocols is a permissionless and easily verifiable method of random sampling one of the nodes that are currently running the consensus protocol. Two skeptical (but related) questions should come immediately to mind: (i) sample from which distribution over nodes, exactly? (ii) in any case, how would you do that without knowing which nodes are running the protocol? Let's put these questions aside briefly and outline how such a subroutine that would allow us to translate our permissioned protocols to the permissionless setting.

BFT-type consensus. Consider the Tendermint protocol, for example. Recall from Chapter 7 that, in each round of the protocol, a single leader node proposes a (new or inherited) block and then the entire set of nodes casts and tracks (two stages of) votes for that round's block proposal. The idea is to use, in each round of the protocol, the assumed method of random sampling a node once to select that round's leader as well as $s - 1$ additional times to select the rest of the "committee" of s nodes (here s is a parameter of the protocol, like $s = 20$ or $s = 100$, and is independent of the actual and unknown number n of nodes running the protocol). The s randomly selected nodes are then the nodes tasked with carrying out the current round of the Tendermint protocol. (If a node is randomly selected more than once for the committee, then each time it was selected, it gets one distinct vote.) In effect, this approach uses the assumed random sampling method as a "wrapper" to reduce permissionless consensus to permissioned consensus.

Longest-chain consensus. The utility of a random sampling method is even more obvious in the case of longest-chain consensus, where each block is proposed unilaterally by a round's leader and nobody votes (other than the implicit votes in each leader's decision of which previous block to extend). Here, there's no need to select a committee. The assumed random sampling method can be invoked once in each round to select that round's leader (corresponding to step (2a) of the pseudocode in chapter 8), who then unilaterally proposes blocks and their predecessors (step (2b)).

9.2.2 Challenge: Sybils

Should a permissionless and verifiable method of random sampling even exist? For example, how can you select a node uniformly at random without knowing the set of possible candidates?

As an initial (bad) idea, imagine that we tried to implement uniformly random node sampling as follows:

(i) nodes register their public keys in a smart contract stored on the blockchain;

(ii) whenever needed, the protocol selects one of the currently registered keys uniformly at random. (Technically, the protocol would select one of the registered keys pseudorandomly, for instance, using the lower-order bits of the hash of some seed. If nodes running the protocol know what hash function and seed the protocol is using, then they will also automatically know which of the registered public keys was selected.)

For example, if the method is used to select a block proposer for a round, then the other nodes know to ignore any block proposals for that round that are not signed with the private key corresponding to the selected registered public key.

The glaring issue behind this approach is that, while it's all fine and good to maintain a list of public keys, the protocol still has no idea about the mapping from registered public keys to actual physical nodes. A single Byzantine node could costlessly create thousands or millions of public-private key pairs (just type ssh-keygen at a Unix prompt) and register them all, in effect, masquerading as a huge number of distinct nodes. (By contrast, you can think of the PKI assumption in a permissioned setting as asserting that there is a one-to-one correspondence between the public keys registered with the protocol and the nodes that are running the protocol.) Then, if there were (say) 99 honest nodes that each registered only once, the one Byzantine node would be selected by the protocol as the block proposer almost every round!

Generally, in computer science, when someone talks about a Sybil attack, they usually mean the manipulation of some protocol through the creation of lots of identities. Basically, a single party masquerades as many. Sybil attacks show up in lots of parts of computer science, not just in blockchains, but I hope it's very obvious why Sybil attacks are a real danger for any random sampling approach to permissionless consensus. In general, deliberately creating multiple identities in a system is often called a Sybil attack. Systems in which Sybil attacks do not help the attacker are called Sybilproof or Sybil-resistant. The naive approach to random sampling above is not Sybil-resistant. This does not, of course, automatically mean that all approaches to random sampling are likewise doomed to fail. Could there be a Sybil-resistant method for randomly selecting one of the nodes running a consensus protocol—a method where the probability of a node's selection is independent of how many identities they might hide behind?
The answer, while not at all obvious, turns out to be "yes." There are multiple methods for Sybil-resistant random sampling, and in this chapter series we'll focus on the two that are most widely used as of 2022:

Note:-

Dominant Approaches to Sybil-Resistant Random Sampling

1. Proof-of-work. (See Section 9.4 for details.) This approach samples a node running the protocol with probability proportional to the total amount of computational power that it contributes.
2. Proof-of-stake. (See Chapter 12 for details.) This approach samples a node running the protocol with probability proportional to the total amount of cryptocurrency that it has locked up in a designated smart contract.

The point is that the quantity that governs a node's probability of selection—the combined computational power of all its identities, or the combined staked cryptocurrencies of all its identities is independent of how many identities it has. This is the sense in which proof-of-work and proof-of-stake are sybil-resistance methods.

9.3 Consensus Protocols vs. Sybil-Resistance Mechanisms

We are not the same. We've talked at length about two types of consensus protocols (BFT-type and longest-chain), and this chapter and chapter 12 will talk about two approaches to Sybil-resistance (proof-of-work and proof-of-stake). Consensus protocols and Sybil-resistance mechanisms are very different concepts, and you should keep them separate in your mind. Conflating them is a common and confusing mistake and will mark you as a newbie to the technical foundations of blockchains.

For example, think back to the permissioned setting with the PKI assumption that we focused on in chapters 2–8. Sybil-resistance is trivial in this setting, each node is simply assumed to have a unique public key and no Sybil-resistance method was needed. On the other hand, we had a lot to say about the various consensus protocols you could use in this setting. In the permissionless setting, you now have two design decisions to worry

about: (i) what underlying consensus protocol to use? (ii) how to prevent Sybil attacks?

Said differently, the purpose of a blockchain's consensus protocol is to decide, given the blocks that have been proposed (and perhaps voted upon) by the protocol's participants, which blocks should be considered finalized (and in what order). As we've seen, there are different approaches to this decision. In a BFT-type protocol, a block is finalized if and only if it is accompanied by an appropriate quorum certificate. However, in a longest-chain protocol, the block is finalized if and only if it is sufficiently deep on the longest chain. These two types of protocols are very different, but both exist for the same purpose: to decide which blocks have been finalized.

Sybil-resistance mechanisms address a different question—who has the privilege of proposing and voting on blocks in the first place? Again, this took care of itself in the permissioned setting (with the PKI assumption), where the protocol could hard-code the allowable voters in a BFT-type protocol (namely, all permissioned nodes) and the method of leader selection in a BFT-type or longest-chain protocol (e.g., round-robin or uniformly at random).

Note:-

Consensus vs. Sybil-Resistance

1. Sybil-resistance. Decide which nodes have the privilege of participating, at which times and in what roles.[In the permissioned + PKI setting, hard-coded into the protocol.]
2. Consensus. Give the block proposals (and possibly votes) of the participating nodes, which ones should be considered finalized?

For example, a common rookie mistake is referred to as "proof-of-stake consensus." This phrase doesn't typecheck, because "proof-of-stake" refers to a sybil-resistance mechanism, not a consensus protocol. It does make sense to speak about a proof-of-stake blockchain, meaning a permissionless blockchain protocol that uses proof-of-stake for sybil-resistance (and something else for consensus). Proof-of-stake blockchains come in many flavors—many use BFT-type consensus, a few use longest-chain consensus, and some use still other approaches to consensus.

Pairing proof-of-work with longest-chain. The discussion above makes it crystal clear that consensus protocols and sybil-resistance mechanisms are two very different things. Accordingly, if we focus on the two dominant methods of consensus (BFT-type and longest-chain) and the two dominant methods of sybil-resistance (proof-of-work and proof-of-stake), we can mix and match to get four different possibilities (Figure 9.1). Many but not all of the leading blockchain protocols can be categorized as one of these four types. Some use alternative approaches to consensus (e.g., Avalanche) and others use alternative approaches to sybil-resistance (e.g., Chia). Are any of the four possibilities better or more natural than the others?

It's become increasingly clear over the past few years that there is a fairly natural pairing between the two approaches to consensus and the two approaches to sybil-resistance. For proof-of-work, Nakamoto got it exactly right in the original Bitcoin protocol: it pairs perfectly with longest-chain consensus and is largely incompatible with BFT-type consensus protocols. (The basic issue is that undetectable fluctuations in the total computational power devoted to the protocol can cause liveness failures, even in the synchronous setting.) Thus, given that Nakamoto had decided to use proof-of-work to overcome Sybil attacks, they had no choice but to invent a new approach to consensus. You sometimes hear the combination of longest-chain consensus with proof-of-work sybil-resistance referred to as Nakamoto consensus, and we'll get into the details of it starting in the next section. As of this writing, the only major blockchain protocols that use Nakamoto consensus are Bitcoin and its forks (like Bitcoin Cash, Litecoin, and Dogecoin).

Pairing proof-of-stake with BFT-type consensus. The story for proof-of-stake Sybil resistance is more nuanced. Most initial experiments with proof-of-stake attempted to use it as a drop-in replacement for the proof-of-work part of the Bitcoin protocol. As we'll discuss in detail in Chapter 12, switching to proof-of-stake sybil-resistance causes a surprising number of complications in longest-chain protocols. It's not impossible to pull off a proof-of-stake longest-chain protocol. There's no impossibility result akin to the one in Section 12 for proof-of-work BFT-type protocols—but it's a lot messier than you'd think. As of this writing, the biggest (by market cap) proof-of-stake longest-chain blockchain protocol is Cardano.

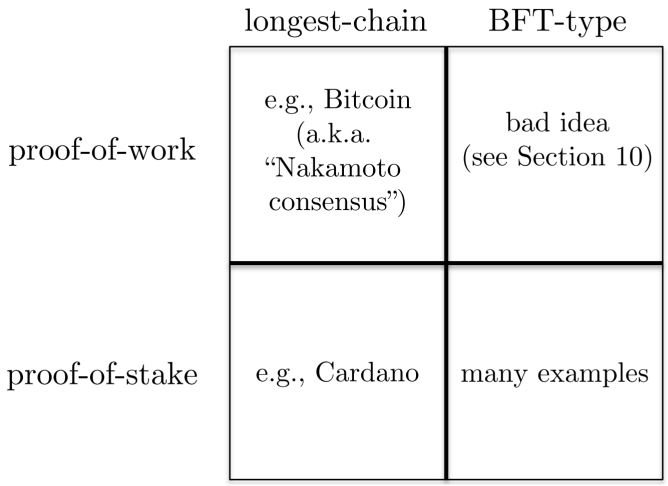


Figure 9.1: The two most common approaches to consensus and to sybil-resistance combine for four categories that together capture many (but not all) of the leading "layer-one" blockchain protocols.

The challenges of proof-of-stake longest-chain consensus design, along with the promise of deterministic rather than probabilistic finality, have led to a shift in recent years toward proof-of-stake BFT-type protocols (which at the time of this writing is the most common of the four combinations among major "layer-one" protocols).

When you think about it, BFT-type protocols couple quite naturally with proof-of-stake sybil-resistance. Arguably the most straightforward way to implement proof-of-stake-based protocol participation is to require participants (identified by their public keys) to lock up capital (in the form of the blockchain's native cryptocurrency) in a designated smart contract for a prescribed period of time. Once the participating public keys are known and publicly visible in the staking contract, we are more or less in the permissioned setting (with public keys playing the roles of nodes) with the PKI assumption. One can then run a BFT-type protocol in which the participating "nodes" are the registered public keys; the voting power of a registered public key should be proportionate to its stake so that no one can artificially and costlessly inflate their control over the protocol by creating a large number of public keys. We'll talk much more about the details and nuances of this approach in chapter 12.

9.4 Proof-of-Work and Nakamoto Consensus

For us, the main point of proof-of-work sybil-resistance will be to extend the permissioned version of longest-chain consensus and its guarantees in chapter 8 to an analog ("Nakamoto consensus") in the permissionless setting.

9.4.1 chapter 8 Recap

We concluded Chapter 8 by isolating the properties of longest-chain consensus that were really driving the analysis, and it's worth recapping those here. Recall from that chapter that the abstract description of longest-chain consensus includes an underspecified leader selection step (step (2a)), in effect assuming that some mapping from rounds to rounds' leaders is carried out by some black box (Figure 9.2). The first step is to start with a genesis block, which is known to all nodes at and only at the time of the protocol's commencement (a trusted setup assumption, called "$a1$" in Lecture 8). The protocol proceeds in rounds and in each round, there is a single leader node (selected in step "$a2$" . In Step "$b2$", the leader of the round proposes blocks and explicit predecessors for those blocks. Honest nodes are instructed to propose a single block that extends the longest chain that they are aware of, breaking ties arbitrarily.

Key properties for the analysis. Three properties of this black box were necessary for the proofs of the

consistency and liveness properties in Chapter 8, none of which are fundamental to the permissioned setting per se:

Note:-

Required Properties of the Leader Selection Box

1. (Same as assumption (A2) from chapter 8) It is easy for all nodes to verify whether a given node is the leader of a given round.
2. (Same as assumption (A3) from chapter 8) No node can influence the probability with which it is selected as the leader of a round.
3. (Required hypothesis for the common prefix property, liveness, and chain quality guarantees in chapter 8) The sequence of leaders is sufficiently balanced. (In longest-chain consensus, all honest nodes are interchangeable (they all behave identically) and all Byzantine nodes are interchangeable (they are colluding anyway). Thus all that matters about a leader sequence is its pattern of H's and A's.)

"Sufficiently balanced" means different things, depending on the context (see Chapter 8 for details), but there's a single sufficient condition that guarantees (with high probability) all of the variants that we care about:

Note:-

Key Sufficient Condition for Consistency and Liveness of Longest-Chain Analysis

Leaders in different rounds are chosen independently and in each round, the probability that the leader is Byzantine is at most $\alpha < \frac{1}{2}$.

In other words, if you push the button on the black box to generate a new leader, the leader that pops out is more likely to be honest than Byzantine (with each selection independent of previous ones). This condition drove all the "proportional representation" arguments that were used throughout Chapter 8, and given that the condition holds the consequent arguments do not depend at all on being in the permissioned setting.

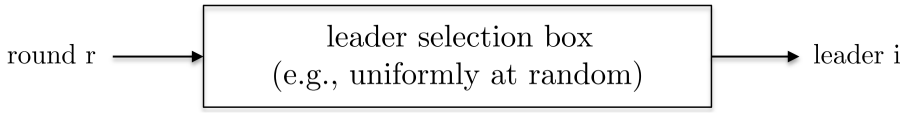


Figure 9.2: Step (2a) of longest-chain consensus is effectively a black box that maps round numbers to leader nodes.

The missing ingredient. All the properties of longest-chain consensus that we care about thus boil down to having a sufficiently balanced leader sequence, and this, in turn, boils down to make sure that the key sufficient condition above holds. In the permissioned setting with the PKI assumption, our solution was to assume that less than half the nodes are Byzantine and to choose leaders independently and uniformly at random. For a permissionless solution, we'll need a different method of (sybil-resistant) random sampling (i.e., proof-of-work) and a different assumption (that less than half of the total computational power is contributed by Byzantine nodes). But as long as we manage to somehow satisfy the sufficient condition above in the permissionless setting, we'll be good to go, with all the consistency and liveness guarantees from Chapter 8 carrying over (assuming, as usual, that the security parameter k controlling the depth-til-finalization is chosen appropriately large).

In summary, the protocol and analysis of Chapter 8 is missing one and only one ingredient from a permissionless protocol with the same consistency and liveness guarantees: a permissionless leader selection box that, under an appropriate assumption about the resources of the Byzantine nodes, selects honest leaders more frequently than Byzantine ones. Section 9.4.2 next will supply exactly this missing ingredient.

Nakamoto consensus vs. abstract longest-chain consensus. As an advance warning, three aspects of the proof-of-work version of longest-chain consensus may surprise you, given the abstract version that we've been

focused on thus far. First, steps (2a) and (2b)— the choice of a leader and that leader's choices of blocks and block predecessors—will be smooshed together into a single step, in effect with each node making its step (2b) choices before knowing whether it's selected as the round's leader. Second, as we'll see in Section 9.4.3, the way the two steps will be smooshed together will prevent any leader from ever proposing more than one block in a single round (i.e., as promised, assumption "$A4$" from chapter 8 will be satisfied). Finally, you might be thinking of each round as having some fixed duration like 10 seconds. In the proof-of-work version of longest-chain consensus, new rounds are triggered by random events and hence, a round's duration is a random variable. By contrast, none of these three idiosyncrasies are present in proof-of-stake versions of longestchain consensus (see Chapter 12).

9.4.2 What Is Proof-of-Work?

'**The basic idea.** So how does "proof-of-work" work? The basic idea is to declare the leader of the next round to be the first node that manages to come up with a solution to a hard puzzle. It should seem plausible that, for an appropriate choice of puzzle, your likelihood of being the first solver of the puzzle should depend on the total amount of computational effort that you put into solving it (with the number of public keys that you control totally irrelevant).

Even at this vague level of description, you should be able to see what we meant in chapter 8 when we stated that, in the proof-of-work version of longest-chain consensus, rounds are "event-driven" (with the event being that some node successfully solves a hard puzzle). Likewise, you see why (as mentioned above) rounds will have variable and random durations (depending on whether some node gets lucky and finds a puzzle solution quicklyor not, or if all nodes are unlucky and take a long time to find a puzzle solution).

In accordance with the permissionless setting and an unknown set of nodes, this approach to randomly sampling a leader (with the leader whichever node happens to get lucky first) is "bottom-up" (implemented by the nodes themselves through a process external to the protocol) rather than "top-down" (implemented in protocol, as in the permissioned version of longest-chain consensus).

A little history. Proof-of-work actually dates back to 1992, seventeen years before the Bitcoin protocol dropped. Dwork and Naor introduced the idea in the context of spam fighting, in order to ak the sending of an email modestly computationally expensive (e.g., taking 0.1 second or 1 second per email on a commodity machine). This was a pretty prescient paper, not only because blockchains didn't exist yet, but (believe it or not) spam email didn't really exist yet either! Nakamoto and also Back's later use of the idea in HashCash was the first to apply the proof-of-work technique in the design of consensus protocols. As of 2022, permissionless consensus is clearly the killer application of proof-of-work.

Cryptographic hash functions. What do I mean by a "hard puzzle", the Saturday New York Times crossword? One could imagine various approaches, but there's one approach that works really well. The same approach taken by Nakamoto in the original Bitcoin paper, where the puzzle is to approximately invert what's called a "cryptographic hash function." To this point in the chapter series, we've adopted only one cryptographic assumption, the relatively uncontroversial assumption that secure digital signature schemes exist (see Chapter 1 for extended discussion). It's possible to sign a message (using a private key) such that anyone can verify the signature (using the corresponding public key), and such that it's impossible (for all practical purposes) to forge such signatures without knowledge of the private key. As we've seen (e.g., in Tendermint), such signature schemes are very useful in the design of (permissioned) consensus protocols (especially when all nodes' public keys are common knowledge at the start of the protocol, a.k.a. the PKI assumption).

Now we're making a second cryptographic assumption—another one that is relatively uncontroversial in practice—namely that cryptographic hash functions (CHFs) exist. (5In this chapter, we'll use CHFs only to implement proof-of-work sybil-resistance. Later in this course, they will be indispensable tools for a totally different reason, in the construction of cryptographic commitment schemes (Merkle trees, etc.).) A hash function, remember, is just a function from some domain to some range. Depending on the context, hash functions can be used to compress or expand the input. In the context of key-value stores implements as hash tables, the point of a hash function is to spread data evenly throughout the hash table's array (cf., "universal hashing").

Cryptographic hash functions are designed with a different goal in mind, which is to be totally inscrutable, meaning (for all practical purposes) totally unpredictable. Ideally, whatever you feed in as input, you get back as output some intelligible gibberish that you've never seen before in your life.

If you want to remember just one function that is believed to be cryptographic in this sense, remember "SHA-

256." (Here "SHA" stands for "secure hash algorithm" and the "256" refers to the number of bits of output. The input to SHA-256 can be of any length.) I encourage you to look up a reference implementation of SHA-256 on the Web—it's not that long, maybe 100 lines of C code. Despite being a deterministic function whose source code is known to everybody, it's a totally inscrutable function. For all practical purposes, the output of SHA-256 is completely unpredictable for any input that you haven't already evaluated it on.

The random oracle assumption. We'll formalize the idea of an inscrutable and unpredictable cryptographic hash function via the random oracle assumption. This assumption will on the one hand be patently false, but in other ways a surprisingly close approximation of reality. The assumption states that the hash function we're using (such as SHA-256) may as well be a random function, in the sense that no program that anyone writes will ever be able to tell the difference. Here by "random function" we mean literally a function chosen uniformly at random from the set of all functions with the given domain and range; or, equivalently, a function for which the output of each input is chosen independently and uniformly at random from the range (like $\{0,1\}^{256}$).
I encourage you to think about a random function as being defined by lazy evaluation, on a need-to-know basis. Think of it as a box with a little gnome in it. When you evaluate the hash function h for the first time on some input x, the gnome in the box flips a coin 256 (say) times and records and outputs the resulting string of 256 zeroes (heads) and ones (tails).

If you ever re-evaluate the hash function at the same input x, the gnome in the box checks its records and produces the same output $h(x)$ as before. But if you feed in some different input y, the gnome in the box will flip a fresh set of 256 coins, recording and outputting the resulting 256-bit string $h(y)$. It should be clear that such a function is completely unpredictable—the only way to learn anything about its output on a given input is to evaluate it on that input, and learning the outputs corresponding to a bunch of inputs tells you nothing about the function's outputs for other inputs.
The assumption that some concrete deterministic function is the same as a random function is of course patently false. SHA-256, for example, is 100 lines of code, not a gnome in a box. But here's the thing: deterministic phenomena can appear random when there is a bounded amount of computational power. For example, imagine you're watching the coin toss at the start of a sporting event, and you try to guess the outcome of the toss while the coin is spinning rapidly while six feet up in the air. It's hard to imagine how you would predict the actual outcome with better-than-50% probability. On the other hand, if you had a much more computationally powerful system than just your naked eye, tons of cameras tracking the coin from different angles, a super-computer dedicated to fine-grained physical simulations, etc.. It's conceivable that you could predict the outcome of the coin flip close to 100% of the time. Similarly, while SHA-256 is in principle completely predictable to a sufficiently powerful (but likely unrealizable) computer, it may well appear random to any computation that we can realistically carry out.

There is also some chance that someone will come up with a new and practical algorithm for better-than-random prediction of a concrete function like SHA-256, and such an event would likely have detrimental consequences for proof-of-work blockchains like Bitcoin. Of course, there's also some chance that someone will come up with an efficient algorithm for the factoring or discrete logarithm problems, in which case all hell will break loose (at least in the short term). Just as we've been happy to assume that there's no fast factoring or discrete logarithm algorithm right around the corner, so too will we assume that better-than-random prediction of SHA-256 won't happen anytime soon. This assumption grows more battle-tested as the years roll by, and this is often all you can hope for when it comes to cryptographic assumptions.

From CHFs to proof-of-work puzzles. Now let's take on faith that we have a concrete function h like SHA-256 which is practically indistinguishable from a random function. How can we use such a function to define hard puzzles? The canonical way is, given a "difficulty threshold" τ :

Note:-

Canonical Proof-of-Work Puzzle
find an input x such that $h(x) \leqslant \tau$.

For concreteness, assume that the hash function h produces 256-bit outputs, which we then interpret as nonnegative integers (written in binary).
The parameter τ is a tunable difficulty parameter. The hardest version of this puzzle is when $\tau = 0$, in which case the task is to invert the function h at the all-zeroes output. The easiest version is when $\tau = 2^{256}$, in which case every input qualifies as a solution. If $\tau = 2^{1}76$,for example, the puzzle is to find an input x such that h(x) has at

least $(256 - 176) = 80$ leading zeroes. In general, bigger τ 's mean easier puzzles (i.e., with more x's qualifying as solutions), and smaller τ's mean harder puzzles. The fact that these types of puzzles have such precisely tunable difficulty is a big part of why they form the basis for all proof-of-work blockchain protocols. For an NP-hard problem like satisfiability or the traveling salesman problem, for example, it's not at all clear what it would mean to "double the instance difficulty." (Whereas with the puzzles above, this would mean cutting the value of τ in half.)

The optimality of repeated guessing under the random oracle assumption. Let's tie this section's threads together by investigating the "partial CHF inversion" puzzles above under the random oracle assumption. Suppose, for example, that we're working with SHA256 and a difficulty threshold of 2^176. The puzzle is then to find an input x such that the output of SHA-256 starts with 80 zeroes in a row.

Now let's adopt the random oracle assumption, and model SHA-256 as a random function h—a gnome in a box. The game is now to find an input x such that $h(x)$ starts with 80 zeroes—i.e., such that the first 80 coin flips by the gnome come up heads. Each time h is queried on a new input, there's a one in 2^{80} chance that the first 80 coin flips are heads. That is, the probability that any given input happens to be a puzzle solution is 2^{-80}. Moreover, under the random oracle assumption, the only way you can learn anything about h is through querying it at various inputs, and its outputs so far tell you nothing about its outputs on as-yet-unevaluated inputs.

What this means is that, in searching for a puzzle solution, there are no options available other than repeatedly evaluating the hash function on some sequence of inputs $x_1, x_2, x_3, \cdots$ until you get lucky and stumble on an input x_i with $h(x_i) \leqslant \tau$. (You can think of repeatedly throwing darts at a dartboard that has a small bullseye.) And because each input is equally likely to be a puzzle solution (e.g., probability 2^{-80}), your probability of success is the same no matter what sequence of inputs you choose to evaluate (as long as they are distinct).

With the parameter choices in our running example, it's hard to find a puzzle solution. With each guess having only a 2^{-80} of succeeding, you would expect to find a puzzle solution only after trying 2^{80} different guesses (of course, it may be much more or much less, depending on your luck). This would take forever on conventional computers, and the only hope of ever finding a puzzle solution would be to throw an enormous amount of specialized hardware at the problem (to explore the input space in a massively parallel way). If τ were 2^{226}, and so a success probability of 2^{-30} per guess, a commodity laptop could find a puzzle solution without too much trouble.

To summarize: (i) if we assume that SHA-256 is indistinguishable from a random function, the only way to find a puzzle solution is through repeated guesses of distinct and otherwise arbitrary inputs; (ii) if each guess is a puzzle solution with probability p, then on average $\frac{1}{p}$ guesses are required to identify a solution. (Technically, this is that fact that the expected value of a geometric random variable with parameter p (i.e., the number of coin flips requires to get a "head," when the probability of heads is p) is $\frac{1}{p}$)

Whatever your doubts about the random oracle assumption, the upshot of the argument above—that in a proof-of-work blockchain, nodes running the protocol will find puzzle solutions via naive repeated guessing is a remarkably accurate description of current practice.

9.4.3 What Should Puzzle Solutions Encode?

We now understand that, in the proof-of-work version of longest-chain consensus, the leader of a round (step "$a2$" in the abstract description) is defined as the first node to find a puzzle solution, meaning identify an input x such that $h(x) \leqslant \tau$, where h is a cryptographic hash function (like SHA-256) and τ is a difficulty parameter (like 2^{176}) that is somehow determined by the protocol.

Committing to block proposals and predecessors. Next, let's drill down on the requirements for a puzzle solution x. Thus far, we've only required that it hashes to a sufficiently small number, but nothing's stopping us from imposing some additional conditions, for example requiring x to conform to a prescribed format. Specifically, it will be convenient to require x to encode a node's choices of blocks and predecessors in step "$b2$" should the node happen to be elected as the next leader—that is, should x happen to hash to a number that's τ or less. Thus, a node will learn whether it's the leader of the next round only after it has committed to its step "$b2$" choices and packaged them into an input x, under the random oracle assumption, the node can't predict whether $h(x) \leqslant \tau$ until it has actually evaluated h on x. (This is the "smooshing together" of steps "$a2$" and "$b2$" of longest-chain consensus that was advertised in Section 9.4.1.) In the other versions of longest-chain consensus (the permissioned version in Chapter 8 and the proof-of-stake version in Chapter 12), by contrast, a leader is generally free to choose blocks and predecessors after it has been selected as the leader of a round.

Forcing a single block proposal and predecessor. But why stop there? In the abstract description of longest-chain consensus, a round's leader is free to propose multiple blocks. (For example, a Byzantine node could propose one block to some of the honest nodes and a conflicting block to the remaining honest nodes.) Meanwhile, the honest leaders are instructed to propose only a single block (extending the longest chain). Given that we're already forcing a node to commit to its step (2b) choices in any candidate puzzle solution, why not automatically disqualify any proposed input x that names more than one block proposal? In other words, let's deem an input x a puzzle solution if and only: (i) x encodes a single block proposal and predecessor; and (ii) $h(x) \leqslant \tau$. This makes it impossible for a leader (Byzantine or otherwise) to propose multiple blocks in a single round. (This is the third comment from Section 9.4.1, promising that the stronger assumption "$a4$" from Chapter 8 would hold for Nakamoto's consensus.) Further proposals can be made only in future rounds in which the same node is again the round's leader. That requires solving an entirely new puzzle; under the random oracle assumption, solving one puzzle provides no advantage to solving a different one.

Format for puzzle solutions. Conceptually, we can think of a candidate puzzle solution x as having multiple fields. Two of the fields should contain a block proposal B and a predecessor pred. You can also think of there being a third field that contains the node's public key "pk" (or at least, one of its public keys) so that the node can claim credit for the puzzle solution. (The motivation for this will become obvious next chapter when we introduce cryptocurrency rewards for block production. In practice, the public key of a round leader is included in its block only indirectly, via what's called a "coinbase transaction.")

Summarizing, the tentative proposal to require inputs x of the form $B||pred||pk$, where "||" denotes concatenation. In addition to satisfying these formatting requirements, an input x qualifies as a puzzle solution only if its hashes to something small (i.e., $h(x) \leqslant \tau$). We would therefore expect a motivated node running the protocol to experiment with different x's, i.e. with different blocks and/or different predecessors, and/or different public keys that it can costlessly generate in its quest for a (properly formatted) input that with a sufficiently small hash. Such experimentation is sometimes called grinding (as a node is grinding through many candidate solutions, looking for a valid one). Given nodes' inevitable experimentation with different x's, it's convenient to include a fourth field called a "nonce" (which stands for "number used once"). The nonce is just a bunch of free bits with no purpose other than providing nodes with degrees of freedom for generating lots of different well-formatted inputs (without needing to, for example, grind over public-private key pairs).

Note:-

Format for Puzzle Solutions
$B||pred||pk||nonce$

Your mental model for a node running a proof-of-work blockchain protocol should be that it fixes a block B of transactions, a predecessor pred for that block, its public key pk, and then grinds through nonces nonce until $h(B||pred||pk||nonce) \leqslant \tau$. (If it hears about a new block in the meantime, produced by some other node, it will likely update its choices of B and $pred$ and restart the nonce grinding process.) Technically, in proof-of-work blockchains like Bitcoin, one often differentiates between the nodes that are grinding away and striving to be selected as a leader and to propose a new block (usually called miners, as if digging for gold (i.e., block rewards)) and the nodes that are merely maintaining the blockchain and checking that miners are following the protocol's rules (usually called full nodes or validators). For example, if you're building on top of a blockchain and want to query the blockchain's state from a smart contract, you care about talking to a validator, not a miner. For simplicity, we're currently assuming that every node running the protocol is acting as both a miner and a validator. This is a surprisingly accurate description of how proof-of-work blockchains like Bitcoin work in practice.

9.5 Properties of Proof-of-Work Nakamoto Consensus

Now that we understand how proof-of-work works, we can return to fulfilling our earlier promises. This section proves that proof-of-work is indeed resistant to sybil attacks (under the random oracle assumption), that it provides the missing "permissionless black box" needed to extend longest-chain consensus from the permissioned to the permissionless model, and notes some additional remarkable properties possessed by Nakamoto consensus.

9.5.1 Sybil-Resistance

The most important property of proof-of-work for our purposes is its sybil-resistance, meaning that the probability that a node is selected as the next leader, i.e. is the first to solve a hard puzzle, or specifically to partially invert a cryptographic hash function is independent of the number of public keys that it might control.
For the analysis, we will adopt the random oracle assumption of Section 9.4.2—that whatever hash function we're using (e.g., SHA-256) is indistinguishable from a random function, in which a gnome in a box flips a fresh set of 256 coins every time a new input is evaluated. We will also assume that every node running the protocol repeatedly tries different *nonces* until some get lucky and finds a nonce that (coupled with the node's choice of the block, predecessor, and public key) hashes to a number that is at most the current difficulty threshold τ. 23 As discussed in Section 9.4.2, under the random oracle assumption, this is the obvious and optimal strategy for the nodes running the protocol.
Under these assumptions, the probability that a given node is selected as the next leader depends only on the total amount of computational power that it contributes to the protocol, independent of the number of identities it might control. Formally:

Theorem 9.5.1 Sybil-Resistance of Proof-of-Work

Suppose n nodes $1, 2, \cdots, n$ make $\mu_1, \mu_2, \cdots, \mu_n$ distinct nonce guesses per second. Then, for every node i and round r,

$$\mathbf{Pr}[\text{node } i \text{ is the round-}r \text{ leader}] = \underbrace{\frac{\mu_i}{\sum_{j=1}^{n} \mu_j}}_{i\text{'s fraction of the overall hashrate}} \qquad (1)$$

Moreover, the selections of leaders in different rounds are independent.

The proof of Theorem 9.5.1 is just some easy probability that follows from the random oracle assumption, as we'll see below. Before that, a few comments. First, the μ_i's are often referred to as the nodes' hash rates, reflecting the fact that we care about a node's "computational power" only since we care about the rate at which it can try out different nonces (which one hash function evaluation necessary and sufficient to check if a given choice yields a valid puzzle solution). (For instance, the specialized hardware (ASICs, for "application-specific integrated circuits") used for Bitcoin mining are not general-purpose computers and are optimized to evaluate SHA-256 as quickly as possible (at the expense of all other functionality).) Second, while the statement of Theorem 9.5.1 references a set of n nodes for notational purposes, we stress that the proof-of-work process works in the permissionless setting and so is independent of what this node set might be. At any given time, there will be some set of nodes running the proof-of-work protocol and searching for puzzle solutions, those nodes will have some hash rates at that time, and Theorem 9.5.1 will then apply to those particular nodes and hash rates at that time. By "node," we mean an actual physical machine running the protocol, not a specific public key. For example, if a node is capable of evaluating one billion nonces per second, this will be true whether all the guesses are concentrated under a single pubic key or split 50/50 between two different public keys. Spinning up multiple threads under different identities doesn't magically increase the number of guesses that the machine can make. Finally, because the right-hand side of (1) is independent of how many identities node i controls, Theorem 9.5.1 implies that proof-of-work is indeed sybil-resistant.

Corollary 9.5.1 Sybil-Resistance of Proof-of-Work

The probability that a node is selected as a leader depends only on its share of the overall hashrate, independent of its number of identities.

Looking at Theorem 9.5.1 and Corollary 9.5.1, it should be clear that a node i can increase its probability of selection by beefing up its machine and increasing its hash rate μ_i (assuming other nodes' hash rates stay fixed, this would increase node i's share of the overall hash rate). But at least this is a costly manipulation (more computers $\Rightarrow$ *moremoney*), *unlikesybilattacks*, *whichareessentiallycostless*(*repeatedlyenterssh*−*keygenataUnixprompt*).*Youmigh*
proofnessinapermissionlesssetting.*Theanalysis*.*Theproof*−*of*−*worksybil*−*proofnessproperty*(*Theorem*9.5.1)*mayseem*

Proof of Theorem 9.5.1: Fix an arbitrary node $i \in \{1, 2, \cdots, n\}$. Consider two events (i.e., things that might or might not happen): event A, that a given guess was made by node i; and event B, that a given guess turns out to

be a valid puzzle solution (and therefore kicks off a new round). The theorem statement concerns the conditional probability $\mathbf{Pr}[A|B]$—the probability that the guesser of a puzzle solution (i.e., the leader of round) is the node i.

Under the random assumption, every guess made by a node produces a fresh string of perfectly random bits (the coin flips by the gnome in the box). This means that the probability that any given guess is a valid puzzle solution is the same no matter which node proposed it. (As all nodes face the same difficulty parameter τ .) In particular, event B is independent of event A: $\mathbf{Pr}[B|A] = \mathbf{Pr}[B]$. Independence of events is symmetric, to event A must then be independent of event B: $\mathbf{Pr}[A|B] = \mathbf{Pr}[A]$. (Formally, this is a special case of Bayes' rule.) The probability $\mathbf{Pr}[A]$ that the current guess was made by node i equals the relative frequency of guesses made by node i, which is $\frac{\mu_i}{\sum_{j=1}^{n} \mu_j}$, as promised.

Finally, under the random oracle assumption, puzzle-solving efforts from past rounds have no bearing on future rounds (with all nodes continuing to repeatedly try out different guesses, each equally likely to succeed). Thus, leaders of different rounds are selected independently.

9.5.2 Permissionless Consensus with Provable Guarantees

Supplying the missing ingredient. Now let's segue from studying proof-of-work in its own right and pick up the story where we left off at the end of Section 9.4.1, examining the composition of proof-of-work sybil-resistance with longest-chain consensus (i.e., Nakamoto consensus). Back in that section, we agreed that the key to the analysis in Chapter 8 (which established several probabilistic consistency and liveness properties for longest-consensus) was the property that leaders in different rounds are chosen independently and, in each round, the probability that the leader is Byzantine is at most $\alpha < \frac{1}{2}$. This property drove all the "proportional representation arguments" that implied (with high probability) various balancedness properties of the generated leader sequence, and these balancedness properties in turn implied the consistency and liveness guarantees (provided the security parameter k is chosen sufficiently large, given the bound α on the probability of a Byzantine leader and the duration of interest). We barely used the power of the permissioned model in these arguments only for implementing the leader selection black box, e.g. using a round-robin order or uniformly random leaders.

In Nakamoto consensus, rounds' leaders are chosen using proof-of-work (with the leader of the next round the first node to partially invert a cryptographic hash function, and with their proposed block and predecessor encoded in the solution). Proof-of-work supplies the missing permissionless ingredient and turns longest-chain consensus is to a purely permissionless consensus protocol—the protocol operates without any knowledge of the set of nodes (with the nodes themselves rather than the protocol selecting leaders, in effect) and with all communication via broadcast. The sybil-resistance property of proof-of-work (Theorem 9.5.1) implies that the probability that a given node is the leader of a given round equals its share of the overall hash rate, with leaders of different rounds chosen independently. By extension, the probability that the leader of a given round is a Byzantine node equals the fraction of the overall hash rate that is controlled by such nodes. Thus, as long as this fraction α is less than $\frac{1}{2}$, we should be good to go! (This ¡ 50% Byzantine hash rate assumption replaces our old assumption from the permissioned model of ¡ 50% nodes.)

Checking the assumptions. Some chores remain before we can declare victory with a permissionless consensus protocol with provable guarantees. Specifically, the analysis of longest-chain assumption relied on five assumptions (above and beyond the permissioned model and the PKI assumption), and we need to check that our proof-of-work implementation of longest-chain consensus doesn't violate any of them so that that chapter's analysis continues to apply to Nakamoto consensus. For reference, here's a restatement of those assumptions from Chapter 8:

Note:-

Assumptions (A1)–(A5)

(A1) No node has knowledge of the genesis block prior to the deployment of the protocol.
(A2) It is easy for all nodes to verify whether a given node is the leader of a given round.
(A3) No node can influence the probability with which it is selected as the leader of a round in step (2a).
(A4) Every block produced by the round-r leader must claim as its predecessor some block that belongs to a previous round.
(A5) At all times, all honest nodes know about the exact same set of blocks (and predecessors).

Assumption (A1). Let's take them one at a time. Assumption (A1) was important in our proof in chapter 8 of the common prefix property. It is a trusted setup assumption. Basically, that Byzantine nodes couldn't get

started on puzzle-solving until the protocol was deployed, meaning that it's the responsibility of whoever deploys the protocol rather than that of the protocol itself. Thus, in our analysis, we simply assume that it's true without worrying about why.
That said, in practice, when deploying a protocol, one can take pains to build up confidence among participants that the trusted assumptions are indeed true. Nakamoto was well aware of Bitcoin's trusted setup assumption, and released the protocol (on January 3rd, 2009) with a genesis block that included a hard-coded message that referenced a very recent headline of the Financial Times (about a bailout for British banks—remember this was in the middle of the Great Recession). Presumably, no one (not even Nakamoto) knew what the Bitcoin protocol's genesis block was until at most a few hours before its deployment.

Assumption (A2). The next three assumptions are asserted properties of the protocol (as opposed to an external-to-protocol trusted setup assumption), we need to check that all of them hold. Happily, with proof-of-work sybil-resistance and with puzzle solutions encoding block proposals and predecessors, these all take care of themselves. Let's start with assumption (A2). A round's leader is defined as having a valid puzzle solution. If a node can exhibit such a solution (an input x that is formatted correctly and with $h(x) \leqslant \tau$), all the other nodes can check its formatting and check for itself that it hashes to something small. (The hash function h is part of the protocol's description and designed to be easy to evaluate, and as we'll see in Section 9.6, any node following along with the protocol knows what the current difficulty parameter τ is.) Thus if you possess such a puzzle solution, all other nodes can easily recognize you as the next leader. If you don't, there's no way to trick the other nodes into thinking that you are the next leader (whatever you try to supply to them will either be improperly formatted or hash to something bigger than τ, so you'll be caught red-handed by the other nodes).

Assumption (A3). Assumption (A3) follows immediately from the sybil-resistance property of proof-of-work (Theorem 9.5.1, which relies on the random oracle assumption). The probability that a node is selected as a round's leader equals its fraction of the overall hash rate, and there's nothing that the node can do about it. (Other than investing in additional hashing power, which occurs at a slower timescale than the one we're analyzing here; see also the discussion following Corollary 9.5.1. This assumption (A3) might more accurately state that no node can costlessly influence the probability with which it is selected as the leader of a round.)
Assumption (A4). For assumption (A4), as advertised, Nakamoto consensus satisfies the stronger condition (A4') that at most one block is produced in each round. This follows from the fact that there is exactly one puzzle solution per round (by the definition of a round) and that a valid puzzle solution can (by definition) name only a single block proposal (see Section 9.4.3). Also, such a puzzle solution can only name as a predecessor a block that existed at the time of its creation, which must necessarily have been produced in some previous round.

Assumption (A5). The final assumption (A5) is an assumption about the communication network (that communication is instantaneous), not the protocol, and so we're not in a position to enforce it. The good news is that this assumption isolated the most difficult aspect of consistency in longest-chain consensus which is finality, also known as the consistency of a node with its future self (as opposed to consistency between nodes at a given moment in time, which is trivialized by (A5)). The bad news is that this assumption is false, even in a well-functioning network, messages have nonzero delays. We'd therefore like to relax this assumption to something more plausible—at the very least, to the synchronous model with some nonzero maximum message delay Δ—without materially changing the consistency and liveness guarantees that we've come to expect for longest-chain consensus. And this is exactly what we'll do in Section 9.7 (under the assumption that Δ is small relative to the expected duration of a round)!

Nakamoto consensus: the final scorecard. Summarizing, here are the key assumptions under which Nakamoto consensus has provable (probabilistic) consistency and liveness guarantees:

Note:-

Key Assumptions for Consistency and Liveness of Nakamoto Consensus

1. The security parameter k is sufficiently large.
2. The random oracle assumption—e.g., SHA-256 is for all practical purposes indistinguishable from a random function (see Section 9.4.2).
3. The trusted setup assumption (A1), no advance knowledge of the genesis block.

4. Less than half of the hash rate is controlled by Byzantine nodes (by Theorem 9.5.1, this guarantees that the key sufficient condition from Chapter 8 is satisfied). [Actually, in the general synchronous model this degrades slightly; see Section 9.7.4]

5. The communication network conforms to the synchronous model, with finite maximum message delay Δ.

6. The difficulty parameter τ is tuned so that the average duration of a round (i.e., the average amount of time between two successive valid puzzle solutions) is much larger than Δ. [How much larger? Think 1–2 orders of magnitude. The precise bound on the maximum-allowable fraction of Byzantine hash rate depends on the ratio between the average round duration and Δ, and approaches 50% as this ratio approaches infinity (see Section 9.7 for details).]

The next two sections (Sections 9.5.3 and 9.5.4) discuss some additional remarkable properties of Nakamoto consensus; the time-constrained reader can skip to Section 9.6 without any loss of continuity.

9.5.3 How Did We Elude the PSL-FLM Impossibility Result?

Strong permissionlessness. While Nakamoto consensus does require a trusted setup assumption—assumption (A1), that no node has advanced knowledge of the protocol's genesis block—it does not require our usual trusted setup assumption, the PKI assumption. That is, nodes running the protocol require no knowledge of the public keys possessed by the other nodes running the protocol. For example, to join the set of nodes running the Bitcoin protocol, all you need to do is download some software, sync up your machine to the current state of the blockchain, and start following along (maintaining the blockchain, trying to find puzzle solutions, etc.). No one else knows that you exist!

While there's no single widely accepted definition of a "permissionless" protocol, no matter what definition you use, the Nakamoto consensus certainly qualifies. At any moment in time, a new block might well get produced under a public key that nobody has ever seen before. (Once that block is published, everybody sees the public key responsible for it. But whoever published the block can generate and switch to using a new public key immediately, thereby returning to puzzle-solving in complete obscurity.) As we'll see in Chapter 12, blockchains based on proof-of-stake sybil-resistance tend to be "somewhat less permissionless," in that the set of public keys eligible for producing the next block is usually maintained in public view (i.e., the set can be derived from the blockchain's current state).

Review: the PSL-FLM impossibility Result. If you've been following this chapter series closely, you might at this point have a question, namely: Why doesn't Nakamoto consensus contradicts the impossibility results for SMR protocols that we saw earlier in the series.

Back in chapter 2, we studied the Dolev-Strong protocol, which solves the Byzantine broadcast problem (satisfying termination, validity, and agreement) even when 99% of the nodes are Byzantine, under two key assumptions: (i) the synchronous model; (ii) the PKI assumption. (Iterating this protocol then gave us an SMR protocol satisfying consistency and liveness.) Later chapters probed to what extent assumptions (i) and (ii) are necessary for this result. For example, in chapter 6 we saw that, in the partially synchronous model, there's no solution to the Byzantine broadcast problem when at least one-third of the nodes are Byzantine (even with the PKI assumption). Relevant here, though, is our probing of assumption (ii) in Chapter 3. Let's flashback the thought experiment on the hexagon and the various ways that a Byzantine node might simulate simultaneously four fictional honest nodes. It stated that without the PKI assumption, even in the synchronous model and even assuming that cryptography exists, no protocol solves the Byzantine broadcast problem when at least one-third of the nodes are Byzantine. Thus, the advance distribution of nodes' public keys was an unavoidable requirement of the Dolev-Strong protocol.

Meanwhile, Nakamoto consensus can solve the state machine replication (SMR) problem even when more than one-third (up to 49%) of the hash rate is Byzantine. Why doesn't this guarantee (without PKI, consensus is possible with 49% Byzantine) contradicts the PSL-FLM impossibility result (without PKI, consensus is impossible with 34% Byzantine)?

Some false resolutions. The answer is subtle, so let's step through some incorrect answers before arriving at the correct resolution.

Guess #1: the PSL-FLM impossibility result only limits the fraction of nodes that are Byzantine, while the Nakamoto consensus guarantees require that less than half the hash rate is controlled by Byzantine nodes.

The "node versus hash rate" distinction is irrelevant. Our guarantees for Nakamoto consensus hold for any distribution of hash rates (as long as less than half of the total is controlled by Byzantine nodes), so they hold in particular when every node happens to have the same hash rate. In this special case, the "¡ 50% hash rate" assumption becomes a "¡ 50% nodes" assumption, which would seem to contradict the one-third threshold established by the PSL-FLM impossibility result.

Guess #2: the PSL-FLM impossibility result is for the Byzantine broadcast problem, while the Nakamoto consensus protocol solves the SMR problem.

This is a better guess, but still incorrect. The hexagon argument from Chapter 3 can in fact be extended to hold also for the Byzantine agreement and SMR problems (see the Appendix of that chapter). Alternatively, Nakamoto consensus can be extended to solve the Byzantine agreement problem when 49% of nodes/hash rate are Byzantine (even without the PKI assumption).

Guess #3: the PSL-FLM impossibility result is for deterministic protocols that guarantee safety and liveness, while the Nakamoto consensus protocol is randomized and guarantees only probabilistic safety and liveness.

It's another good guess, but again not the correct resolution. As it turns out (an excellent tricky homework problem), the proof of the PSL-FLM impossibility result in chapter 3 can be extended to rule out randomized protocols for Byzantine broadcast (or agreement) that achieve both safety and liveness with high probability. (Whereas Nakamoto consensus does guarantee both with high probability, as long as the security parameter k is sufficiently large.)

Guess #4: the new trusted setup assumption (A1) breaks the proof of the PSL-FLM impossibility result.

All the proof of the PSL-FLM impossibility result requires is for a Byzantine node to be capable of simulating the behavior of four honest nodes. Assuming that there's a genesis block that is unknown in advance does not at all interfere with such simulations.

Guess #5: the random oracle assumption breaks the proof of the PSL-FLM impossibility result. As with the previous guess, the assumed existence of a random oracle (accessible to both honest and Byzantine nodes) does nothing to prevent Byzantine nodes from simulating sets of fictitious honest nodes.

The actual explanation. The real reason is:
Byzantine nodes are strictly less powerful in our model of Nakamoto consensus than in the standard permissioned model.

If true, this would resolve the seeming contradiction between the guarantees of Nakamoto consensus and the PSL-FLM impossibility result: the former works with a weaker type of adversary and thus can tolerate a higher fraction of them than would otherwise be possible. Remember assumption (A4), the stronger version of (A4) satisfied specifically by the proof-of-work version of longest-chain consensus? In our proof-of-work model, a Byzantine leader is restricted to propose at most one block in its round. This restriction has no analog in the standard permissioned model, and it rules out (among other things) the canonical ploy of a Byzantine node, which would to be fabricate two conflicting blocks (both belonging to the same round), tell some honest nodes about one of them and the rest about the other one. The guarantees provided by Nakamoto consensus prove that taking away this and similar Byzantine strategies fundamentally weaken the power of Byzantine nodes, thereby turning the impossible into the possible.

To better appreciate this point, I encourage you to revisit our analysis of the permissioned version of longest-chain consensus in Chapter 8. In that chapter, the PKI assumption allowed us to easily satisfy assumptions (A2) and (A3) about the verifiability of leader selection. The leader of each round was common knowledge (e.g., derived from a shared global clock or pseudorandomness derived from the blockchain state) and, because of the PKI assumption, a round's leader and only a round's leader was in a position to make (appropriately signed)

block proposals in that round. This prevented Byzantine nodes from, for example, masquerading as leaders out of turn or spoofing block proposals by leaders of previous rounds. If you drop the PKI assumption, you'll find that any attempt to rework longest-chain consensus breaks down because of the enlarged space of feasible Byzantine strategies. In particular, you'll find it difficult to enforce assumption (A4), that every proposed block can only name as a predecessor a block that was first created in an earlier round. (And of course, we know from the PSL-FLM impossibility result that any such attempt must break down.)

In Nakamoto consensus, proof-of-work leader selection and assumption (A4') render these additional Byzantine strategies impossible, despite the lack of a PKI assumption. When we implement longest-chain consensus with proof-of-stake sybil-resistance in chapter 12, Byzantine nodes will be free to make multiple block proposals in the same round and we'll need to go back to making the PKI assumption (which will in any case be quite natural in that context).

Where does the PSL-FLM proof break? The preceding discussion explains why the guarantees of Nakamoto consensus don't automatically contradict the PSL-FLM impossibility result— proof-of-work sybil-resistance effectively restricts the strategies that are otherwise available to Byzantine nodes. To complement this understanding, let's revisit the proof of the PSL-FLM impossibility result (from Chapter 3) and see how it breaks down in the proof-of-work setting.

The proof there posited a hexagon of honest nodes with inconsistent initialization files. There were three cases, with each case corresponding to a bona fide instance of Byzantine broadcast with two honest nodes (who acts as in the hexagon) and one Byzantine node (who simulates the joint behavior of the remaining four hexagon nodes, thereby tricking the two honest nodes to behave exactly as they would in the hexagon). The three cases (along with the termination, validity, and agreement requirements) imposed a contradictory set of three conditions on the behavior of any correct protocol, thereby showing that no such protocol can exist.

In all our chapters on permissioned consensus (chapters 2–8), we didn't worry about nodes' computational power. We assumed that honest nodes were powerful enough to carry out whatever a protocol asked of them, and that Byzantine nodes could do whatever they want (other than break cryptography). In particular, in the proof in Chapter 3, we didn't blink an eye when some Byzantine node needed to simulate a collection of four (fictitious) honest nodes from the hexagon. In that context, the Byzantine nodes couldn't possibly do 4 times as much computational work as things the honest nodes are expected to do.

In the proof-of-work setting, however, we're making a fine-grained assumption about the power of Byzantine nodes (namely, that their collective hash rate is less than the collective hash rate of the honest nodes). And simulating another node's hash rate requires having that hash rate yourself. For example, if all nodes (honest and Byzantine) have the same amount of hash rate, a Byzantine node could simulate one honest node but not four. The node can't magically fabricate four times as much hash rate as it actually has. In other words, because proof-of-work makes simulation costly, Byzantine nodes can no longer carry out the "quadruple-simulation strategies" that are needed to push through the proof of the PSL-FLM impossibility result. And as the guarantees of Nakamoto consensus show, this breakdown of the proof is fundamental and cannot be salvaged by some different and more clever argument.

9.5.4 Does Nakamoto Consensus Need a Shared Global Clock?

To what extent does our analysis of Nakamoto consensus require the nodes to agree upon a common notion of time? We are working in the synchronous model, which by default comes equipped with a shared global clock. But does the Nakamoto consensus ever use it? In the version we've been discussing thus far (in which the difficulty parameter τ magically falls from the sky), the answer is no: Rounds are defined in an event-driven way (as opposed to by the passage of time), so nodes must agree only on valid puzzle solutions and not necessarily on any common notion of time. This is different from all the other consensus protocols (based on longest-chain or otherwise) that we'll study, which require nodes to have (at least approximately) synchronized clocks. (For example, this assumption will be important for the proof-of-stake longest-chain protocols that we discuss in Chapter 12, which produce blocks on a deterministic schedule.)

Unfortunately, as we'll see in Section 9.6, Nakamoto consensus does need to rely on a notion of time to implement a "difficulty adjustment" algorithm that is responsible for tuning the difficulty parameter τ to target a particular rate of block production. However, even with difficulty adjustment, Nakamoto consensus tends to be more forgiving of loosely synchronized clocks than other consensus protocols (longest-chain or otherwise).

9.6 Difficulty Adjustment

Note: this section (on difficulty adjustment) and the next two sections (on extensions to the standard synchronous model and an impossibility result for proof-of-work BFT-type protocols, respectively) can be read in any order.

9.6.1 The Opposing Forces on the Difficulty Parameter τ

Proof-of-work puzzles are parameterized by a difficulty threshold τ , with large values of τ corresponding to easier puzzles (as a solution in an input x that hashes to a number that is at most τ). How, exactly, should you set τ in a proof-of-work blockchain protocol?

The faster the block rate, the better? Relatedly, you may have noticed that the value of the parameter τ seems to have played no role in our consistency and liveness analysis (Section 9.5.2)! This should seem like a red flag, so let's think through what's going on. What, intuitively, are the pros and cons of setting τ larger or smaller? As we increase τ, puzzles get easier, and thus the rate of solution-finding (and hence block creation) by nodes increases. Seems like a pure win, For one, the time-to-finalization (also known as "latency"), the time necessary for a block on the longest chain to be extended k times decreases if the time between blocks decreases. (Remember from Chapter 8 that the security parameter k depends on the fraction of Byzantine hash rate, the duration of interest, and the tolerable probability of a consistency or liveness violation. We have not yet seen any reason why k would depend on the choice of τ, so twice the rate of block creation would seem to mean half the time to finalization.) For two, more frequent blocks (assuming there's some fixed maximum block size) should mean more throughput, meaning more transactions processed per second.

Inadvertent honest forks. The flaw in this line of reasoning is that it implicitly assumes that every block that gets created will wind up on the longest chain (as in Figure 9.3(a)). The worry is that if puzzles are too easy and blocks are created too frequently, then an increasing share of the created blocks will be orphaned, off of the longest chain. Because remember, there are two different reasons that forks can show up in longest-chain consensus. The first is deliberate forks caused by Byzantine nodes. The second is inadvertent forms caused by honest nodes. But honest nodes always extend the longest chain, you say, so how did they ever create a fork? An honest node can only extend the end of the longest chain that it knows about; if an honest node's information is stale, the longest-chain that it knows about may not be the longest chain overall, and by extending the former it causes a fork. In its simplest form: imagine that two honest nodes are both dutifully trying to extend the current block B of the longest chain, the first node succeeds and creates a block B_1 that extends B, and very soon thereafter (before the first node can communicate B_1 to the second node) the second node also succeeds and creates a block B_2 that extends B. The two honest nodes inadvertently created a fork in the blockchain (as in Figure 9.3(b)), and at most one of the blocks B_1 and B_2 can wind up on the longest chain and at least one of them will be orphaned.

(a) (b)

Figure 9.3

Honest nodes are vulnerable to inadvertent forks if the time it takes for them to communicate can exceed the length of a round (with the second node succeeding before hearing about the first node's success). We haven't had to discuss such forks thus far in this chapter, but that's an artifact of our convenient but unrealistic assumption (A5) that we're working in the super-synchronous/instant-communication model. In this model, all honest nodes effectively communicate by telepathy and learn instantaneously about any newly created blocks by honest nodes and immediately switch to trying to extend the latest one. Thus, inadvertent honest forks are impossible in this model, no matter how quickly blocks are created (i.e., no matter how easy the puzzles are, no matter how large τ is).

Summary. The difficulty parameter τ should be set to balance two competing goals: (i) a reasonably fast block rate; and (ii) a reasonably low frequency of inadvertent forks caused by honest nodes on account of outdated information. Working with assumption (A5) suppressed consideration (ii), which is why the value of τ has not mattered in any of our analyses so far. Section 9.7 relaxes assumption (A5) and extends all of our provable guarantees for longest-chain consensus to the standard synchronous model (with some known maximum message delay Δ). In that section, we'll learn that, in order to keep the rate of inadvertent honest forks under control (and thereby preserve our consistency and liveness arguments), the difficulty parameter τ should be set so that the typical length of a round (i.e., the typical time between successive puzzle solutions) is larger than Δ by a constant factor (e.g., somewhere between 5 and 100, depending on how conservative you want to be). So, how should the parameter τ be set?

Note:-

How to Set τ

Set the difficulty parameter τ to target a desired rate of block production, which in turn should be a constant factor larger than the maximum message delay.

One approach to tuning the constant would be to set a maximum tolerable orphan rate (e.g., 10%) and take τ (and hence the block rate) as large as possible subject to this constraint.

9.6.2 Case Study: Bitcoin

Nakamoto anticipated that the amount of hash rate participating in the Bitcoin protocol could vary widely over time (due to Moore's law, the increasing popularity of the protocol, etc.) and, famously, that protocol tunes its difficulty adjustment parameter τ to target the production of on average one block per ten minutes (6 blocks an hour, 144 blocks a day, 1008 blocks a week). This is a relatively conservative target, perhaps two orders of magnitude larger than typical message delays over the Internet.

One, just because the popularity of the protocol could vary over time, but also because of technological advancements, Moore's law, etc. That would also be a force causing hash rates to increase as time goes on. Hopefully, it's intuitively clear that if you want to keep the rates of block production, in other words, the frequency with which puzzles get solved, if you want that to remain roughly constant, even as the hash rate fluctuates, well, you better make sure that the threshold tau fluctuates in the same way. To maintain this target rate of block production, the Bitcoin protocol must adjust the difficulty parameter to offset changes in the hash rate (e.g., if the hash rate doubles, τ should be cut in half). In mode detail, whenever 2016 new blocks have been added to the longest change, the protocol updates τ. Note that 2016 is the target number of blocks for a fortnight (two-week period). So the protocol looks and how long it actually took for the longest chain to grow by this amount. If it took longer than expected (e.g., four weeks instead of two) the puzzle difficulty is adjusted accordingly (e.g., doubling τ). Similarly, if those 2016 blocks were added over the course of only one week, then the protocol would halve the difficulty parameter. In general, if it took $\beta \cdot 14$ days to add 2016 new blocks to the longest chain, the difficulty parameter τ gets multiplied by β. (This formula ensures that, if the hash rate stays the same, then future blocks will be created on average once per 10 minutes. Taking this approach means that τ is a lagging indicator of the overall hash rate; the hash rate has for the most part been increasing over time and as a result, the average rate of block production has generally been a little faster than one per ten minutes)

9.6.3 The Role of Block Timestamps

If you've been paying close attention, you might be puzzled by the preceding description of Bitcoin's difficulty adjustment mechanism. In Section 9.5.4 we mentioned that one remarkable property of Nakamoto consensus, as opposed to the permissioned and proof-of-stake implementations of longest-chain consensus, is its lack of reliance on a global shared clock. This property follows from the event-driven nature of Nakamoto consensus (with rounds delineated by new puzzle solutions), as opposed to the time-driven approach of other longest-chain consensus implementations. Meanwhile, in Bitcoin, the difficulty adjustment algorithm measures the amount of (real world) time that elapsed during the addition of the most recent 2016 blocks on the longest chain. How does the Bitcoin protocol know what time it is? Isn't the Bitcoin blockchain supposed to act as a hermetically sealed environment? The one concession the Bitcoin protocol makes to acknowledging the outside world is that every block is required to include a timestamp (in addition to transactions, a pointer to a predecessor block, and other block metadata), precisely so that the difficulty adjustment algorithm has the information that it needs. A block's timestamp is set by its producer. (Honest) nodes running the protocol are supposed to include accurate timestamps, and also to enforce some rules around timestamps so that they can't be manipulated too much by Byzantine nodes. (Because the timestamps play no role other than measuring the elapsed time every 2016 blocks, minor timestamp manipulations have a negligible effect on the protocol.) The details of these rules are way out in the Bitcoin weeds (and arguably the least elegant aspect of the protocol's design), so for this chapter, we'll just assume that all blocks come with accurate timestamps.

9.6.4 Work-Adjusted Longest-Chain Consensus

The inevitability of and complications from difficulty adjustment. With proof-of-work sybil-resistance, significant fluctuations in the overall hash rate over time seem inevitable, necessitating a difficulty adjustment algorithm to retain control over the rate of block production. While difficulty adjustment may be unavoidable, that doesn't stop it from being annoying and leading to other problems. One example will be the next chapter (chapter 10), in which we'll see how deliberate forking attacks and delayed block creation announcements can manipulate the difficulty adjustment algorithm to boost the share of block rewards earned by a deviating node.

This section concerns a different issue, namely that we now need to revisit what we mean by a longest chain. To this point, "longest chain" has meant among all chains, the one with the largest number of hops (equivalently, the chain whose tip is farthest from the genesis block). And this is indeed the appropriate definition for the permissioned implementation of longest-chain consensus, as well as typical proof-of-stake implementations. For Nakamoto consensus, this would be the appropriate definition only if the overall hash rate stayed fixed (and, generally, it doesn't).

A bad example for naive longest chain. To see the issue, imagine a Byzantine node that has 10% of the collective hash rate of the honest nodes. If all nodes attempt to produce blocks of the same difficulty (i.e., with the same τ), then honest nodes will create blocks 10 times as often as the Byzantine node. If the Byzantine node is only trying to create blocks that are 100 times easier (i.e., with τ 100 times as large as that for the honest nodes), however, then it will create blocks (of low difficulty) 10 times as often as the honest nodes create blocks (of high difficulty). In the most extreme case, the Byzantine node could go back to the genesis block (when the proof-of-work puzzles were incredibly easy) and rapidly create a long chain of super-easy blocks (setting the blocks' timestamps in such a way that difficulty adjustment does not kick in and so puzzles remain super-easy). Were the protocol to use the naive (unweighted) longest-chain rule, the Byzantine node might be able to eventually create an alternative chain (of low-difficulty blocks) that is the longest chain overall, leading to a rollback of a massive number of blocks (a big consistency/finality violation). Various details of the Bitcoin protocol interfere somewhat with this Byzantine strategy, but this simplified example provides accurate intuition for why the protocol should weigh blocks by difficulty as opposed to treating them all the same.

Question 2

Propose a concrete strategy for a node with less than 50% of the overall hash rate that conforms to all of Bitcoin's timestamping rules and leads to a consistency violation (assuming that the security parameter k is 100).

Weighting by the amount of work. There's an obvious fix to this issue, which is to weigh a block according

to the amount of work that went into producing it, and then redefine "longest chain" as the chain with the most overall weight. While we can't know exactly how much work (i.e., how many attempted hashes) went into the production of a block. Only the puzzle solution is recorded, not all the failed attempts that led up to it. We do know the expected number of hashes required to create a block with a given difficulty threshold. For instance, with SHA-256 and a difficulty threshold of 2186, any given attempt has a $2^{186}/2^{256} = 2^{-70}$ chance of succeeding (under the random oracle assumption), which means that in expectation 2^{70} hashes must be attempted before finding a puzzle solution. We can thus define the work of this block to be 2^{70}. More generally, we can define the work of a block as the expected number of attempts that would be required to produce that block. Because the work of a block is a function only of the difficulty threshold τ (which is maintained by the protocol and its difficulty adjustment algorithm throughout its execution) and other hard-coded parameters (e.g., the length in bits of a candidate puzzle solution), the protocol can easily compute the work of each block. (In fact, in the Bitcoin protocol, the work of a block can be read off immediately from its metadata (which includes the current value of τ).) We can then define the work of a chain as the sum of the work in each of the chain's blocks. Honest nodes in Nakamoto consensus are then instructed to propose only blocks that extend the highest-work chain (and, as usual, to include all known outstanding transactions and announce the new block immediately), without regard to the number of blocks in the chain. In the preceding example, a Byzantine node with 10% of the overall hash rate could still create a long alternative chain of blocks with low puzzle difficulty, but because the total amount of work in that chain will be so low, honest nodes will continue to ignore it.

Revisiting consistency and liveness. We proved our consistency and liveness guarantees in chapter 8 (and extended in this chapter) for longest-chain consensus specifically for the case of fixed difficulty and the simple block-counting definition of "longest"? Do these guarantees carry over to general Nakamoto consensus (with variable difficulty and difficulty adjustment) with our work-adjusted version of the longest chain rule?
Happily, the answer is yes, provided the following three (hopefully unsurprising) assumptions hold (in addition to standing assumptions summarized at the end of Section 9.5.2):

Note:-

Additional Assumptions for Consistency and Liveness of Work-Adjusted Longest-Chain Consensus

(i) The security parameter k is sufficiently large.

(ii) At every moment in time, less than half the hash rate is controlled by Byzantine nodes.

(iii) The amount of overall hash rate changes only at a bounded rate over time.

Assumption (i) we've had all along, but we're restating it here to emphasize that, for provable consistency and liveness guarantees, the parameter k may need to be chosen somewhat larger in the variable-difficulty case than suggested by the original fixed-difficulty analysis. Assumption (ii) hopefully strikes you as obviously necessary—as we saw in chapter 8, if Byzantine nodes control 51% of the overall hash rate for a while, they can wreak havoc (e.g., force the rollback of many blocks) through deliberate forking attacks. The interaction between assumptions (i) and (iii). The need for assumption (iii) may not be obvious, so let's talk through an extreme example to develop intuition about it. Suppose you're happily running longest-chain consensus with security parameter $k = 100$ (say), the difficulty has stayed constant so far, and 1000 blocks have been created and added to longest chain. The first 900 blocks on this chain are thus regarded as finalized. But now suppose there's a massive (billion-fold, say) increase in the overall hash rate. Now, all blocks from the "low-difficulty period" will count for almost nothing during the subsequent "high-difficulty" period. For example, if a Byzantine node happens to be the first one to create a high-difficulty block, and that block extends the genesis block, then all the honest nodes (who dutifully extend the chain with the most work) will trigger a consistency violation by abandoning the previous chain (including the 900 once-thought-finalized blocks) in favor of extending the new high-difficulty block.32

Question 3

Flesh out this example further so that it properly accounts for the lag between a hash rate increase and the subsequent difficulty increase.

The role of assumption (iii), then, is to rule out such extreme (and arguably unrealistic) examples. For

example, suppose we're willing to assume that the overall hash rate at most doubles during any given epoch (2016 new longest-chain blocks in Bitcoin). It's still true that blocks produced after the doubling carry more weight than those produced before, but only twice as much. Thus, as long as the longest chain at the time of the doubling has a significant lead over the next-longest chain, it will still have a meaningful (though less secure) lead after the doubling. This argument shows why the security parameter k should be set larger in the variable-difficulty setting than in the fixed-difficulty setting: a larger k means that rolling back k blocks is highly unlikely even after the overall hash rate doubles and pre-existing blocks count only half as much. The wilder the fluctuations in hash rate that are allowed in assumption (iii), the bigger the increase in the security parameter k that is needed to compensate.

This concludes our heuristic argument that under assumptions (i)–(iii) and those at the end of Section 9.5.2, variable-difficulty Nakamoto consensus satisfies consistency and liveness (with high probability) even when 49% of the hash rate is controlled by Byzantine nodes. For a fully rigorous analysis, making precise the intuition described here, see the paper by Garay et al. .

9.7 Extending the Analysis to the Synchronous Model

Section 9.5.2 reviewed the assumptions under which we proved (in Chapter 8) that longest chain consensus guarantees (probabilistic) consistency and liveness (with at most 49% of the hash rate controlled by Byzantine nodes). Assumption (A1) is a trusted setup assumption (no advance knowledge of the genesis block) and we're continuing to take that on faith. We argued there why Nakamoto consensus satisfies assumptions (A2)–(A4) (under the random oracle assumption). That left us with the assumption (A5), our unrealistic adoption of the "instant communication" or "super-synchronous" model in which honest nodes can communicate instantly, as if by telepathy. In addition to being patently false in any real communication network, this assumption trivialized consistency between different honest nodes. (It did not, you'll recall, trivialize finality, meaning an honest node's consistency with its future self.) we've promised over and over again that all the guarantees we've proved for longest-chain consensus continue to hold (with minor degradation) in the general synchronous model (with an a priori known bound Δ on the maximum message delay), provided the average duration of a round (i.e., time elapsed between consecutive puzzle solutions) is larger than Δ (say, 1-2 orders of magnitude). This section, finally, supplies the math that fulfills this promise. This extension is straightforward when rounds have deterministic durations, as in typical permission and proof-of-stake implementations of longest-chain consensus. The idea is to set the round length bigger than Δ so that at the start of each round honest nodes can be sure to have heard about messages (like announcements of newly created blocks) sent by other honest nodes at the beginning of the previous round. The present setting of Nakamoto consensus, with its variable-length rounds, is the one in which non-trivial work is required. This is also the section where, finally, the analysis will give us concrete guidance about how to the difficulty threshold τ in Nakamoto consensus. (With assumption (A5), τ played no role in our analysis. Here, we'll see that it should be set to target a rate of block production.))

9.7.1 Isolating the Role of Assumption (A5)

Let's review all our results from Chapter 8 about longest-chain consensus and examine exactly where our proofs break down without assumption (A5).

Liveness and chain quality. In chapter 8, under the assumption (A5), we proved that as long as each leader of longest-chain consensus is chosen independently and has a bigger-than-50chance of being an honest node, then the longest chain will (with high probability) contains blocks produced by honest nodes regularly. (Because an honest node includes all outstanding transactions that it knows about in its block, this implies that a transaction known to all honest nodes will eventually be included in a finalized block.)
The proof of this guarantee hinged on the following property:

- Whenever there's a round with an honest leader, the length of the longest chain known to all honest nodes increases by one.

This follows from the fact that all honest nodes know about the same set of blocks (under the assumption (A5)) and that an honest node extends the end of the longest chain that it knows about. The following consequence of

this property is enough for our purposes:

(*) the length of a longest chain known to an honest node is at least the number of blocks that have been produced thus far by honest nodes.

Why was property (*) useful in the proof? Imagine that each leader has a 51% chance of being honest and consider the first 1000 rounds. In expectation, this sequence will have 510 honest leaders and the key property implies that the longest chain will grow in length by at least 510 during this time. Because there were only 490 Byzantine leaders during this time, at least 510 - 490 = 20 of the last 510 blocks on the longest chain must have been contributed by honest nodes. In Nakamoto consensus, the leader of a given round can produce only one block and thus trivially can contribute only one block to the longest chain. Even under the weaker assumption (A4) from Chapter 8, the version appropriate for the permissioned and proof-of-stake implementations of longest-chain consensus, the leader of a given round can produce multiple blocks overall but can contribute only one block to any given chain.

The same argument implies a more general "chain quality" guarantee for longest-chain consensus: if Byzantine nodes control an α fraction of the hash rate, then over T time steps the longest chain grows by at least $(1-\alpha)T$ (in expectation) with at most of the growth due to blocks produced by Byzantine nodes. That is, we expect at least a $\frac{1-2\alpha}{1-\alpha}$ fraction of the blocks on the longest chain to be contributed by honest nodes.

Failure of property (*). Property (*) breaks down when we pass from the super-synchronous model (assumption (A5)) to the standard synchronous model with non-zero message delays (up to a known bound Δ). For example, suppose all nodes are honest and have produced a longest chain ending with the block B. All the nodes then attempt to produce a block extending the block B; at some point, some nodes will succeed in producing such a block B_1. Because the node is honest, it will immediately announce the new block so that other nodes can stop working to extend the tip B of the old longest chain and switch to the tip B_1 of the new longest chain. These announcements might take up to Δ time steps to arrive, however, which raises the possibility that some other node will succeed in extending B with a new block B_2 before it hears B_1's announcement (Figure 9.4). Whenever this happens (as it will, inevitably, on occasion), property (*) is violated: two blocks get produced by honest nodes (like B_1 and B_2) but the longest chain known to honest nodes increases by only one. This example also provides accurate intuition for why this breakdown of property (*) doesn't have to be a big deal. Such honest inadvertent forks occur only if two nodes solve puzzles at almost the same time (less than Δ time units apart). As long as puzzles are typically solved infrequently (on average once every L time step, with L much bigger than Δ), such near-ties should be rare. Rereading the liveness and chain quality arguments above, it would seem that they should go through as long as most (if not all) honestly produced blocks increase the length of the longest chain. As we'll see later in this section, this is indeed the case.

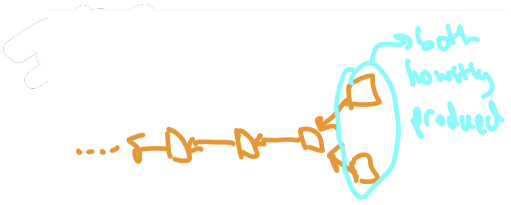


Figure 9.4

The common prefix property. The analysis in Chapter 8 also leaned on the assumption (A5) in its proof of the common prefix property, the property that (for a suitably large choice of the security parameter k) all the longest chains known to an honest node agree except perhaps on their last k blocks. This analysis hinged on the property:

(**) at each block height, there is at most one block produced by an honest node.

Why was this property useful? Our proof of the common prefix property in Chapter 8 proceeded by contradiction, and showed that every violation of the common prefix property must be caused by an unbalanced leader sequence (with Byzantine leaders outnumbering honest leaders in some long window), which in turn happens

only rarely when at least 51% of the hash rate is controlled by honest nodes. This contradiction rested on a counting argument that established a lower bound on the number of Byzantine leaders and an upper bound on the number of honest leaders in a long window. Property (**) gave us the latter bound, which implied in turn the former bound. This was the argument specifically for Nakamoto consensus, using the stronger version (A4') of assumption (A4) stating that each leader can only produce one block. As discussed in Chapter 8, a more complicated analysis is required for permissioned and proof-of-stake implementations of longest-chain consensus, which satisfy (A4) but not (A4').

Failure of property (**). A glance at Figure 9.4 shows that property (**) also breaks down when passing from the super-synchronous model to the standard synchronous model with non-zero message delays—if one honest node has stale information and is unaware of a block recently produced by some other honest node, it might wind up creating a second block at the same block height. Again, this breakdown doesn't seem a big deal as long as such inadvertent honestly created forks are rare (which, intuitively, should be the case if the average time L between puzzle solutions is much bigger than the maximum message delay Δ). Presumably, the counting argument still works (with mildly degraded parameters) if almost all of the blocks produced by honest nodes have distinct block heights?

Finality. The last guarantee that we proved in Chapter 8 was (probabilistic) finality, meaning that with high probability, blocks that are at least k blocks deep on the longest chain will never be rolled back (for a sufficiently large security parameter k). (As always with longest chain consensus, the most recently added blocks on the longest chain should be viewed as tentative and still under negotiation.) If we look back at the argument in Chapter 8, we'll see that assumption (A5) played no role, the only way to violate finality is to violate the common prefix property. Thus, as long as we're able to repair the proof of the common prefix property in the standard synchronous model, we'll immediately get finality in that model for free.

Consistency. Remember that for the state machine replication problem, "consistency" means two things, consistency between different honest nodes at a given moment in time (i.e., no two honest nodes have finalized different blocks at the same block height), and consistency between an honest node and its future self (i.e., "finality"). The former was trivialized by assumption (A5), with all honest nodes automatically in sync with identical information at all moments in time. Thus, in passing to the general synchronous model, we not only have to fix our previous proofs of liveness and the common prefix property, but we are also now responsible for proving consistency in full. Happily, it turns out that consistency follows from the (general version of the) common prefix property in much the same way that finality does, so we'll leave the details of that argument as an exercise for the reader.

Summary. What goes wrong when you relax assumption (A5) and consider the synchronous model with non-zero message delays is the appearance of inadvertent honestly created forks (as in Figure 9.4). The hope is two-fold: first, that such forks are infrequent as long as the typical time L between puzzle solutions is sufficiently larger than the maximum message delay Δ; and second, that as long as such forks are infrequent, our analysis in chapter 8 suffer minimal damage and largely carry over. Next, we provide further details on why both of these hopes are correct.

9.7.2 The Role of Puzzle Difficulty

By now it is evident that two parameters (and the ratio between them) should play a crucial role in our revised analysis:

- Δ, the maximum message delay (in time steps);
- L, the expected number of time steps between successive puzzle solutions.

The parameter Δ is a property of the communication network, which the protocol has no control over. The parameter L, on the other hand, is determined by the protocol's choice of difficulty parameter (and by the current amount of overall hash rate). That is, the protocol is in a position to choose L however it wants, by setting the difficulty threshold τ appropriately.

To explain, suppose the amount of overall hash rate is fixed, with nodes collectively capable of H attempts at puzzle solutions per time step. By the random oracle assumption, each of these attempts has the same probability p of succeeding. (E.g., in the usual case of SHA-256 and $\tau = 2^{186}$, $p = \frac{2^{186}}{2^{256}} = 2^{-70}$.) Then, in each time step, the

probability that a puzzle solution is found (and a new round starts) is well approximated by Hp. For example, if $H = 2^{50}$ and $p = 2^{-70}$, then the chance of some node finding a puzzle solution in any given time step is roughly one in a million. This approximation applies when H is small, in which case the probability that more than one solution is found in the same time step is negligible and can be safely ignored. The story is similar when Hp is not small, just with somewhat less convenient formulas. The expected duration of a round—the expected number of time steps that elapse before a new solution is found—equals the expected number of flips of a coin with bias Hp needed to see "heads." This is a geometric random variable (see footnote 18), with expectation of $1/Hp$. In our running example (with $H = 2^{50}$ and $p = 2^{-70}$), we should expect to see a new puzzle solution on average every million or so time steps. If each time step represents one millisecond, say, then the average duration of a round would be roughly 1000 seconds ($116\frac{2}{3}$ minutes). In general, for the case of an SHA-256-based crypto puzzle, the average duration L of a round satisfies

$$L = \frac{1}{Hp} = \frac{2^{256}}{H.\tau}$$

Thus, for any fixed hash rate H, the protocol can target a desired rate of block production by setting τ appropriately ($\tau = \frac{2^{256}}{HL}$). In other words, once a policy decision has been made about the right choice for the average round length L, the target of the protocol's difficulty adjustment algorithm is uniquely determined. And how should this policy decision be made? As we'll see below, it's wise to choose L to that Δ/L is reasonably small (e.g., at most 0.1). And how should you decide on an appropriate value for the worst-case message delay Δ? That depends on the communication network and its reliability. For a geographically concentrated and well-connected set of nodes, perhaps Δ could be taken as a few hundred milliseconds. For nodes scattered across the globe communicating over the Internet, Δ might be an order of magnitude larger.

9.7.3 Safe Blocks

In Section 9.7.1 we built intuition for the hope that, as long as inadvertent forks by honest nodes (due to two puzzles found at nearly the time) are infrequent, our analysis from Chapter 8 should carry over with minimal loss. The next step is to analyze just how infrequent such forks are, as a function of our two key parameters, the maximum message delay Δ and the average length L of a round. The definition of safe blocks. To motivate the next definition, think about an honest node that successfully finds a puzzle solution at some time step t, resulting in a new block that extends the longest chain that the node knew about at that time. The worry is that the node had stale information about the longest chain and was therefore doing redundant work, creating a fork rather than extending the true longest chain (as in Figure 9.4). Because honest nodes announce newly created blocks immediately and these announcements cannot get delayed more than Δ time steps, this worry can only materialize if some other honest node successfully found a puzzle solution at one of the time steps $t = t-\Delta+1, t-\Delta+2, \cdots, t$. We'll call a block safe if it is produced by an honest node at some time t and it is the only such block produced at any of the time steps $t = t - \Delta + 1, t - \Delta + 2, \cdots, t$. Properties of safe blocks. While properties (*) and (**) don't hold in general in the synchronous model (Figure ??), they do hold for safe blocks (because an honest node is always up to date on safe blocks when it creates a safe block).

Proposition 9.7.1 Honest Blocks Having Increasing Heights

The heights of safe blocks are strictly increasing over time.

Proof: Suppose that safe blocks B_i and B_j are produced by (honest) leaders i and j. There's only one leader per round, so one of i or j is chosen as a leader first—let's say i, with its block B_i at height h. Because i is honest, it announces its block B_i immediately, and this announcement reaches node j at most Δ time steps later. Because there are at least Δ time steps between different safe blocks (by definition), this announcement reaches j prior to the time step in which it creates its block. Thus, because j is honest, B_j will extend one of the longest chains that j is aware of at the time. Because j is aware of a block at height h (namely, B_i) at that time, its block B_j will have height at least $h + 1$. 😊

Because the length of a longest chain equals the maximum height of a block, Proposition 9.7.1 implies that the length of a longest chain is at least as large as the number of safe blocks that have been produced (the analog of property (*), with safe blocks playing the previous role of honestly produced blocks). Proposition 9.7.1 also immediately implies that there is at most one safe block at each block height (the analog of property (**)).

Fraction of blocks that are safe. We can classify blocks into three types: safe (and hence produced by an honest node), produced by an honest node but unsafe, and produced by a Byzantine node. In the analysis below, we're going to play it safe and treat unsafe blocks as if they were produced by Byzantine nodes. Thus, it had better be the case that most of the blocks produced by honest nodes are safe rather than unsafe. The next lemma quantifies the extent to which this is true.

Lenma 9.7.1

$$(1-\alpha).(1-\frac{\Delta}{L})$$

fraction of blocks are safe.

The point is that, as long as L is large relative to Δ, the fraction of safe blocks is almost as large as $1-\alpha$, which was the fraction of honest blocks in our original analysis (under the assumption (A5)). Because properties (*) and (**) hold for safe blocks, that original analysis should hopefully carry over without much trouble (details to follow in the next section). In other words, if Delta's small relative to L, we have the same thing as before basically just with a little bit of loss. The proof, it's not hard. It's just really some simple probability, so what I want you to think about, I want you to think about zooming in on a moment in time where some node, maybe honest, maybe Byzantine, some node comes up with a new puzzle solution. What we're going to be interested in is the probability that that particular puzzle solution results in the creation of a safe block.

Proof of Lemma 9.7.1: Zoom in on a time step t at which some block is created by some node. For this block to be safe, two things much be true: the node producing it must be an honest node, and no other puzzle solutions were found in any of the times steps $t-\Delta+1, t-\Delta+2, \cdots, t$. That is:

Pr[puzzle solution creates safe block] = **Pr**[solution found by honest node]×**Pr**[no puzzle solutions in last Δ time steps]

Under our usual assumptions about proof-of-work (random oracle assumption, with nodes searching for puzzle solutions by brute force), these two events are independent and so no conditioning is necessary.
By the sybil-resistance of proof-of-work (Theorem 9.5.1), we know that the first term on the right-hand side equals the fraction of hash rate controlled by honest nodes (i.e., $1-\alpha$).
What about the second term? Because the average duration of a round is L, and the duration of a round is a geometric random variabl with parameter $\frac{1}{L}$ (see Section 9.7.2)—the probability of a solution in any given round is $\frac{1}{L}$. Thus, the probability of a solution at any of the Δ relevant time steps $\{t-\Delta+1, t-\Delta+2, \cdots, t\}$ is at most $\frac{\Delta}{L}$. This follows by the Union Bound, which states that the probability of the union of a collection of events is no larger than the sum of their individual probabilities. We last encountered the Union Bound in chapter 8, when analyzing the balancedness properties of random leader sequences. The probability of the complementary event which is the second term on the $R.H.S.$ is therefore at least $(1-\frac{\alpha}{L})$.

9.7.4 Updating the Analysis

Analyzing Nakamoto consensus appears harder in the synchronous model than the super synchronous/instant-communication model because there are now three types of blocks rather than two. As usual (by Theorem 9.5.1), we expect an α fraction of the blocks to be produced by Byzantine nodes and a $1-\alpha$ fraction by honest nodes. In the latter group, we expect (by Lemma 9.7.1) unsafe blocks to contribute at most a $(1-\alpha)(\Delta/L)$ fraction of the total blocks and safe blocks at least a $(1-\alpha)(1-(\frac{\Delta}{L}))$ fraction. Unsafe blocks can't be counted on to advance any of our goals, so the simplest and most conservative approach is to treat them as if they were controlled by Byzantine nodes. (A Byzantine node always has the option of acting as if it were honest, so this approach can only make our guarantees stronger.) If you're thinking in terms of the H's and A's of a leader sequence (as in chapter 8), this analysis approach effectively switches some of the H's in the leader sequence to A's (with a switching probability of $\frac{\Delta}{L}$). Our original analysis of Nakamoto consensus (under the assumption (A5)) proved consistency and liveness provided

fraction of blocks by honest nodes ¿ fraction of blocks by Byzantine nodes,

or equivalently provided that $1-\alpha > \alpha$ (i.e., $\alpha < \frac{1}{2}$).

In the general synchronous model, with unsafe blocks effectively switching sides (from honest to Byzantine) to bat for the other team, we can piggyback on that same analysis provided

fraction of safe blocks ¿ fraction of Byzantine blocks + fraction of unsafe blocks.

By Lemma 9.7.1, this inequality holds (in expectation) if

$$(1-\alpha).(1-(\frac{\Delta}{L})) > \alpha + (1-\alpha)(\frac{\Delta}{L})$$

or, equivalently, if

$$\alpha < \frac{1-2\frac{\Delta}{L}}{2-2\frac{\Delta}{L}} \quad (2)$$

Note that when $\Delta = 0$ we recover the $\alpha < \frac{1}{2}$ condition from the super-synchronous case.But more generally, when Δ is small relative to L, the condition degrades only mildly to $\alpha < \frac{1}{2} - \delta$ for some small $\delta > 0$. Thus, provided difficulty is tuned so that L is well larger than Δ, Nakamoto consensus really can tolerate nearly 50% Byzantine hash rate.

Theorem 9.7.1 Nakamoto Consensus in Synchronous Model

Provided the assumptions at the end of Section 9.5.2 and inequality (2) hold, Nakamoto consensus satisfies (probabilistic) consistency and liveness.

To review why Theorem 9.7.1 is true: by (2), safe blocks are produced at a higher rate than Byzantine blocks and unsafe blocks combined. Safe blocks satisfy property (*) in Section 9.7.3, and this property is all that is needed to carry out the liveness analysis from Chapter 8. (The chain quality guarantee from Chapter 8, based on the same argument, also carries over with mildly degraded parameters.) Safe blocks also satisfy property (**) in Section 9.7.3, and this is the main property needed to prove the common prefix property as in Chapter 8. (The revised version of this property for the general synchronous model is: at all times, for every pair i, j of honest nodes, and every choice C_i and C_j of longest chains known to i and j at that time, respectively, the common prefix of C_i and C_j encompasses all but at most the last k blocks of both C_i and C_j .) Finality (and consistency between nodes) follows easily from the common prefix property as in chapter 8.

Interpretation for Bitcoin. Our analysis demystifies why the Bitcoin protocol has a 10-minute block time. Going by the Bitcoin white paper, Nakamoto appeared to be well aware of the importance of the ratio $\frac{\Delta}{L}$ for the protocol's consistency and liveness guarantees. It's not clear what value of Δ Nakamoto may have had in mind for maximum Internet delays on a typical day, but perhaps it could have been something like 10 seconds. Nakamoto suggested tuning the proof-of-work puzzle difficulty so that the parameter L would be 10 minutes. In this case, the ratio $\frac{\Delta}{L}$ would be under 2% and, plugging into the formula in (2), the protocol would satisfy (probabilistic) consistency and liveness even with 49% (if not 49.9%) Byzantine hash rate.

Theorem 9.7.1 also lets us speculate how Bitcoin's guarantees might degrade were we to speed up the rate of block production (e.g., with the goal of boosting throughput). For example, with a block time of one minute (and continuing to assume that Δ is 10 seconds), the ratio $\frac{\Delta}{L}$ would be $\frac{1}{6}$ and, due to an increased rate of inadvertent forks created by honest nodes, the guarantees of Theorem 9.7.1 would apply only when less than 40% of the hash rate is Byzantine. Ratchet up the block rate (i.e., decrease L) further, and still less Byzantine hash rate can be safely tolerated.

9.8 Impossibility of Consistency Under Partial Synchrony

9.8.1 The Impossibility Result and Its Consequences

Pairing proof-of-work sybil-resistance with BFT-type consensus? In Section 9.3 we made a big deal of the fact that sybil-resistance mechanisms (which decide who gets to propose and vote on blocks) and consensus protocols (which decide which of the proposed blocks get finalized and in what order) are fundamentally different

things, and can be paired up in multiple ways. For example, restricting to what are currently the most dominant approaches to sybil-resistance (proof-of-work and proof-of-stake) and blockchain consensus (longest-chain and BFT-type), there are four ways of pairing them together (see the grid in Figure 9.1).[As a reminder, many but not all of the biggest "layer-1" blockchain protocols can be classified as one of these four (or really, three) types.] Nakamoto consensus shows that longest-chain consensus pairs particularly well with proof-of-work sybil-resistance. BFT-type consensus pairs well with proof-of-stake sybilresistance (as we'll elaborate on in Chapter 12), essentially because the type of registration typically required in proof-of-stake protocols enables a reduction from permissionless to permissioned consensus. There have also been successful examples of blockchains that combine proof-of-stake sybil-resistance with longest-chain consensus (such as Cardano). What about the fourth pairing, BFT-type consensus with proof-of-work sybil-resistance?

Extending Nakamoto consensus to the partially synchronous model? A seemingly quite different question is whether we can improve longest-chain consensus so that it has provable guarantees in the partially synchronous setting defined in Chapter 6 (with arbitrary message delays up to an unknown global stabilization time GST, after which the communication network acts as in the synchronous model with all message delays bounded above by an a priori known parameter Δ).
A brief review: BFT-type SMR protocols like Tendermint (chapter 7) typically satisfy consistency (always) and eventual liveness (meaning liveness soon after GST passes). Guaranteeing consistency even under attacks and outages (i.e., pre-GST) is the reason why, in Tendermint, we forced nodes to assemble (two stages of) supermajority quorum certificates before finalizing a block.

With longest-chain consensus, meanwhile, consistency breaks down badly in the partially synchronous model, even when all of the nodes are honest. (we've proven lots of nice guarantees about longest-chain consensus, but they've all been in the synchronous model.) The argument is simple (Figure 9.5): imagine a network partition (in the spirit of the CAP theorem from chapter 6), with two groups of nodes and all intergroup messages delivered immediately and all intragroup messages delayed until GST. The nodes of each group will extend the longest chain known to them, resulting in two parallel chains growing for the duration of the network partition. No matter what the value of the security parameter k in longest-chain consensus, provided the network partition goes on long enough (and remember, GST can be arbitrarily large), and the two groups of nodes will finalize conflicting blocks. This is a violation of consistency, and when the network partition ends, the group of nodes with the shorter longest chain will be forced to roll back several thought-to-be-finalized blocks. Is there any way we can add a little complexity to longest-chain consensus to avoid consistency violations in the partially synchronous model? An impossibility result. Next, we'll prove an impossibility result that shows both why coupling proof-of-work sybil-resistance with BFT-type consensus is a bad idea, and why all Bitcoin-like protocols are doomed to consistency violations in the partially synchronous model. Here's the formal statement:

Theorem 9.8.1 A Blockchain CAP Theorem

Even without any Byzantine nodes, no blockchain with proof-of-work sybil-resistance satisfies both:
(1) liveness in the synchronous setting (even with $\Delta = 0$); and
(2) consistency in the partially synchronous setting.

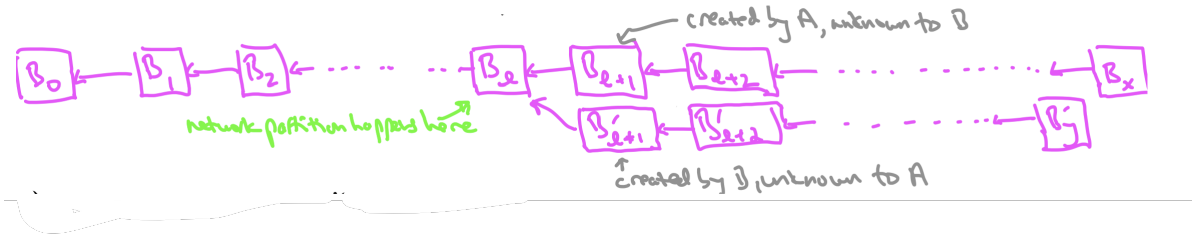


Figure 9.5: With a network partition, the two sides of the partition (group A and group B) independently create two competing chains.

By virtue of using proof-of-work sybil-resistance and satisfying liveness in the synchronous model (Theorem 9.7.1), The Nakamoto consensus must necessarily suffer consistency violations in the partially synchronous model.

(We already knew this via a direct argument; the point here is that knowing nothing about Nakamoto consensus other than that it uses proof-of-work sybil-resistance and is live in the synchronous model is already enough to automatically conclude that it cannot guarantee consistency in the partially synchronous model.) More generally, Theorem 9.1 implies that any modification of Nakamoto consensus (e.g., as seen in Bitcoin) that retains two of its essential properties (proof-of-work sybil-resistance and guaranteed liveness in the synchronous model) is doomed to suffer consistency violations in the partially synchronous model. This answers the second question above in the negative.

As for the first question, imagine that we attempt to couple proof-of-work sybil-resistance with a BFT-type consensus protocol like Tendermint. This could be done in a variety of ways, but let's focus on couplings that preserve Tendermint's guaranteed consistency in the partially synchronous model. For example, one could imagine using proof-of-work sybil-resistance to randomly select a committee of nodes (with sampling probability proportional to hash rate) and task that committee with carrying out some number of rounds of the Tendermint protocol. Assuming the sampled committees are at least two-thirds honest—which would be true (with high probability) provided more than two-thirds of the total hash rate is honest and the target committee size is sufficiently large. This approach would inherit Tendermint's guaranteed consistency in the partially synchronous model. Theorem 9.8.1 would then imply that this seemingly reasonable protocol fails liveness, even in the safe confines of the synchronous model and with no Byzantine nodes. It's hard to imagine deploying a blockchain protocol that doesn't satisfy even the most basic liveness guarantees, so it's not surprising that we have seen any protocols of this type in the wild.

Escaping impossibility via proof-of-stake. We'll see in chapter 12 that Theorem 9.8.1 does not apply to proof-of-stake blockchain protocols; properly implemented, such a protocol can satisfy liveness and consistency guarantees comparable to a permissioned protocol like Tendermint. We'll also see that proof-of-stake blockchain protocols have some other issues. They are in some sense "less permissionless" than proof-of-work protocols and also suffer from some additional attack vectors (such as "long-range attacks").

9.8.2 Informal Proof of Theorem 9.8.1

The one-sentence summary of why Theorem 9.8.1 is true is that, in a proof-of-work blockchain protocol, honest nodes cannot distinguish between a waning hash rate and delayed messages. Note that even without any Byzantine nodes, the (honest) hash rate can fluctuate arbitrarily. Likewise, in the partially synchronous model, messages can be delayed arbitrarily even when there are no Byzantine nodes.

Concretely, zoom in on an honest node i and suppose that, at some point in time, it stops hearing any new messages from any other nodes. Two possible explanations (even without resorting to Byzantine nodes) are:

(i) all other (honest) nodes turned off their machines (i.e., there's no other hash rate in the system);

(ii) it's pre-GST in the partially synchronous model and all messages sent by all other (honest) nodes have been delayed (e.g., due to a network outage).

You have to pity poor node i at this point, as it's stuck in a catch-22 and has only two options: Either it eventually finalizes an additional block, despite not hearing from any other nodes, or it waits (possibly forever) to finalize further blocks until it hears enough messages from other nodes.

If node i chooses the first route (as would be the case in Nakamoto consensus) in order to preserve liveness but the reason for silence happens to be (ii), then the other nodes may well be finalizing their own (conflicting) blocks, resulting in a consistency violation. If node i chooses the other option (as it would if following the Tendermint protocol), a problem arises in scenario (i): with no other nodes to hear from, node i would wait forever and no further blocks would ever be finalized (a violation of liveness, even in the synchronous (and even $\Delta = 0$) model).

Why doesn't the same argument apply to proof-of-stake protocols, in which nodes are sampled with probability proportional to the amount of stake (in the blockchain's native currency) that the nodes have placed in escrow? Because stake is directly observable (recorded right there on the blockchain, in plain view) while hash rate is not (e.g., you cannot definitively deduce the current amount of hash rate from the state of a proof-of-work blockchain protocol). While node i in the proof above can't know if all other nodes suddenly turned off their

machines, it would know immediately if all other nodes suddenly withdrew their stakes. That is, the analog of scenario (i) above can be detected without any communication between nodes in a typical proof-of-stake blockchain protocol, and this fact derails the argument above.

9.9 A Look Ahead to chapters 10–13

You might be surprised that it's possible to write and read a couple of hundred pages about blockchain protocols without ever talking about cryptocurrencies but with chapters 1–9 that's exactly what we've done. And indeed, while the story of blockchains and cryptocurrencies have been tightly intertwined to date (and may continue to be for a long time), cryptocurrencies are not fundamental to blockchain protocols. As we've seen, there can be both permissioned and permissionless blockchain protocols (with provable liveness and consistency guarantees) without any native currency. That said, the next four chapters will focus on blockchain protocols with a native currency, and some of the additional challenges that come up in the design of such protocols.
Endowing a blockchain protocol with a native currency (with the monetary supply and distribution managed by the protocol itself) is convenient for some reasons:

Interesting in its own right. Perhaps the goal is in fact to create a new currency, a type of digital cash that can be used directly for peer-to-peer payments. Why use a blockchain? Because blockchain protocols can be used to track digital ownership without any centralized intermediary (under the usual assumption that a sufficiently large fraction of the nodes/hash rate/stake is running the protocol honestly). It's not obvious that blockchain protocols are the only possible way to achieve this functionality, but they currently have no real competition. It would seem that this goal was Nakamoto's primary motivation for inventing the Bitcoin protocol (what with the white paper's title of "A Peer-to-Peer Electronic Cash System").

Incentive nodes to run the protocol correctly. Even if you're not trying to challenge the US dollar or create a currency in the usual sense, a currency native to a blockchain protocol is useful for some reasons (as the means if not the ends). For starters, in a permissionless blockchain, why should nodes bother to run the protocol at all? There's only so far that a blockchain protocol can scale if relying only on altruistic or curious hobbyists. This question is particularly acute for proof-of-work protocols, given that participation requires devoting a large amount of computation to what is otherwise useless work.

One natural approach is to use incentives, meaning to reward nodes for their participation in something that has real economic value. But what could be in a protocol's possession that could be used as rewards to nodes? If the protocol controls a native currency, then one obvious answer is to pay out rewards in that currency, printing new money if necessary. (Conversely, if the protocol does not control a native currency, there's no obvious solution to the problem.) The protocol is in a position to print new money when needed precisely because the currency is native to it.

Implementing node rewards is particularly straightforward for protocols that use longestchain consensus: every time a new block gets finalized (i.e., sufficiently deep on the longest chain), pay a block reward to the (unique) proposer of that block. For example, the Bitcoin protocol currently pays out 6.25 (newly minted) Bitcoins to the proposer of each finalized block. (Of course, for a reward denominated in the native currency to have meaningful economic value, the currency itself must have some value (e.g., with people e willing to pay a non-trivial amount of fiat currency for it). We won't discuss here the question of when and why a cryptocurrency might have significant economic value.)

If a protocol is doling out economically meaningful block rewards to nodes for producing blocks, it's intuitively clear that some number of people are going to be motivated to participate in the protocol and earn those rewards. But whenever you introduce incentives into a system, you need to take a step back and ask: wait, do the rewards incentivize nodes even more strongly to behave in other, unintended ways?

Chapter 10 is all about answering this question for block rewards in Nakamoto consensus. Perhaps counterintuitively, we'll see in that chapter that, in some settings, honestly following Nakamoto consensus is not the profit-maximizing strategy for a node—by deviating from the intended behavior in clever ways, a node can earn more rewards than it would otherwise!

Charge for usage. A second convenient use of a native currency is to charge for usage. In our chapters thus far, we've been assuming that blocks can be large enough to include all pending transactions known to a node. As you can imagine, in practice, it's necessary to impose a cap on how big blocks can be. If the demand for a blockchain protocol (the number of transactions submitted to it) exceeds its supply (the number of transactions it can process), the protocol must decide which transactions get included and which ones get excluded. As with any over-demanded resource, charging for usage is a natural way to limit demand. (Even if supply exceeds demand and in principle, all transactions can be included without charging anything, nominal transaction fees can still be a good idea (e.g., as an anti-spam measure).)

Implementing this idea requires a new component of a blockchain protocol that we haven't discussed yet, the component that decides which transactions get included and what the creators of those transactions have to pay. This component is known as a transaction fee mechanism and is the subject of Chapter 11.

Proof-of-stake sybil-resistance. As mentioned many times, in chapter 12 we will explore proof-of-stake sybil-resistance, in which nodes are selected with probability proportional to the amount of money they have locked up in escrow (as opposed to the amount of computational power that they contribute). As we'll see, proof-of-stake sybil-resistance has some advantages (and disadvantages) over proof-of-work, and for this reason, has become the dominant approach to sybil-resistance in blockchain protocols over the past five years. A proof-of-stake protocol could in principle accept stakes in some external currency (such as a stablecoin), but it's easier and more practical to require staking using a native currency (as all major proof-of-stake protocols do).

Economic security. Finally, a native currency provides a blockchain protocol with a lever by which to control the amount of security that it provides to its users. By rewarding nodes for participating, a protocol entices nodes to devote economic resources (computational power or capital) to running it. Nodes incur realized or opportunity costs from such investments. The bigger the rewards, the bigger the costs that the nodes running the protocol are willing to absorb.

Chapter 13 studies one definition of the "economic security" of a blockchain protocol, roughly defined as the monetary cost that must be paid by an attacker in order to take over the protocol (e.g., by acquiring lots of hash rate or staking a sufficiently large amount of native currency). We'll see in that chapter that, to first order, economic security scales with the amount of investment by non-Byzantine nodes, which in turn scales with the economic value of the rewards paid to the nodes running the protocol.

Chapter 10

Block Rewards and Selfish Mining

10.1 What we see in this chapter

Chapter 10 marks rather an interesting turning point for us in this course because thus far, we've been focused entirely on the problem of consensus which of course is the fundamental problem that pretty much any blockchain protocol must solve it and must keep a bunch of computers in sync so they all agree on the state of the blockchain. So, accordingly, we've been studying lots of protocols and possibility results thinking about consistent thinking about liveness, and looking at different models of the communication Network and then even in Chapter 9 we started talking about permissionless consensus, civil resistance, proof of work, but it's all been about consensus all the time. There are two things that we now are wondering about:

- The first thing we might be wondering about or started wondering after chapter 9, when we started talking about proof of work and those running a protocol and working hard to try to solve these hard crypto puzzles, we haven't talked out all about incentives. So why would any node bother to run run of these consensus protocols. For example, if you're IBM in the 1980s and you're just buying seven servers to replicate a database for high uptime, who cares you don't need to worry about incentives; the company just buys the seven servers and runs it, end of story! However, in a permissionless context where basically nodes can just start running the protocol whenever they want especially if running the protocol involves doing all this hashing and looking for solutions to hard crypto puzzles, what would motivate a node to bother to join the party and run your consensus protocol?
- The second thing is that we've now had nine chapters but, we haven't talked about cryptocurrencies at all and some of you maybe are wondering that aren't cryptocurrencies the entire point of a blockchain protocol? The answer to that question is not right. So, it's true that the story of cryptocurrencies and the story of blockchain protocols have been very intertwined up to this point and that may well continue to be true for some time but as the last many chapters have shown, you can have a blockchain protocol without having a cryptocurrency and this chapter will be the first one where we're not talking about blockchain protocols completely abstractly, but specifically the protocols that host a cryptocurrency that have some currency native to the protocol. So what do I mean by a native currency? Well, I just mean currency where the money is minted by the blockchain protocol (that's the only way it comes into existence) and then furthermore, once it's in existence, the ownership of the coins that exist is also tracked by the blockchain protocol and its state. now obviously, there are a lot of things one can say about cryptocurrencies in this chapter we're going to focus on quite narrowly, we're going to look at the use of cryptocurrencies to answer the first question that we said might be on your mind: what incentivizes nodes to run a protocol in the first place and one very convenient way to do that is through the idea of block rewards where you reward nodes for contributing to the protocol, so for example, if the nodes are running Nakamoto consensus and trying to find trying to solve hard crypto puzzles for the privilege of publishing the next block every time some node is successful and actually does produce a block that winds up on the longest chain you could imagine, rewarding that node in some way.

  Now, where's that money going to come from? Well, the easiest solution to that is that the reward or the money would come from the protocol itself, that is the rewards would be doled out in the blockchain's

native currency.

So in this chapter, we're going to talk about the pros and cons of motivating nodes to run your protocol through block rewards denominated in a protocol's native currency. we'll see that's already a quite interesting and deep topic in the next few chapters. We'll study some other economic and incentive issues around cryptocurrencies.

In Chapter 11 we'll focus on another convenient use of having a native currency which is you can have transaction fees paid in that native currency so if you have a blockchain protocol usually you want for each transaction you execute, you probably want to charge at least some nominal fees like a penny or something just as an anti-spam device and then if you have very high contention for the blockchain, you may want to charge much more than that just to make sure the most valuable transactions are the ones that you're spending your time on.
That idea of having transaction fees paid for in the blockchain's native currency will lead us to the topic of transaction fee mechanism design and that's what we'll talk about in chapter 11.

In Chapter 12, we'll talk about a totally different convenient use of having a native currency which is an alternative approach to civil resistance so an approach different from proof of work that has some advantages over proof of work known as proof of stake and so that's kind of a simple resistance mechanism that makes sense primarily if you have a blockchain protocol with its own native currency and then chapter 13 that'll be the final chapter about incentive and economic issues around cryptocurrencies and blockchain protocols that one will focus on economic security. So, when someone talks about the cost of launching a 51 attack on a blockchain protocol, in Chapter 13 will talk about how those sorts of n bers get measured and how they connect to the economic value of the blockchain's native currency.

It is worth mentioning that all of these very convenient uses of a native currency all of that is above and beyond the idea that you might just want a cryptocurrency for its right, you literally might want to invent some digital version of cash and that it would seem was Nakamoto's perhaps primary motivation for why they developed the Bitcoin protocol, but the point is that even if you don't care about cryptocurrency, it still has tremendous value just as the means to doing all these different things, incentivizing nodes, running the protocol, charging for the usage of your protocol, proof of stake, civil resistance, and trying to boost the economic security of a blockchain protocol.

10.2 Block Rewards

10.2.1 Why should a node who's free to run a consensus protocol should bother to do that?

Why not? One answer would be why not? Maybe to some of us it seems interesting as if it's cool to cut Edge technology, why not kind of download the latest and greatest blockchain protocol and make your home computer a node and run that protocol. Especially if you think about Nakamoto consensus, if you're doing proof of work civil resistance, the reason why not would be there's a lot of other things you could be doing with all of your computational power, you could be solving other computational problems that are maybe of more intrinsic interest to you or you could be turning your machines off and saving electricity. So especially with the Nakamoto consensus' with proof of work civil resistance, it seems hard to imagine you could just rely on enthusiastic amateurs to run a global blockchain protocol.

So the original answer to this question in Nakamoto's Bitcoin protocol which has been copied by many not all but many subsequent blockchain protocols are to have what's called a block reward. For these rewards to be effective (they need to be economically valuable in order to motivate the nodes to run the protocol) and so the question is where does that money come from? The simplest solution would be maybe the protocol can just Mint new coins, but it's not like new US dollars but it's new coins in the currency it controls (the native currency the blockchain protocol). So whenever it needs to reward a node it mints some new coins in the native currency and then that's what the reward is is paid in.

How it works. It's probably easiest to see how this would work in a longest chain consensus protocol, which is the types of protocols we're going to be focusing on this chapter. Because in longer chain consensus every block is just a unilateral decision made by some nodes by the leader of some round. So to each block that winds up getting finalized, that is each block that winds up sufficiently deep on the longest chain, it's natural to just reward the unique node that produced that block and so that's how it works originally in Bitcoin and most other longest chain protocols. For example, if it's Nakamoto consensus so it's as long as chain plus proof of work civil resistance and nodes are trying to solve these hard crypto puzzles in order to generate a valid block, as soon as you solve that puzzle you've created a block and as long as that block winds up getting included and sufficiently deeply in the longest chain you're going to get your economic reward for all of that hard puzzle solving work that you did.

In many BFT-type protocols such as Tendermint, which we discussed at length in chapter seven, there's still a notion of a block proposer like if you go back to the Tendermint pseudocode you will see that in every round, there's a single leader node who proposes a block for that round which isn't voted upon in a couple of stages by all of the other nodes. You could take this same tack in a BFT-type protocol and if you were the node who made a block proposal that wound up getting finalized at a given block height, it could be that you would then get some direct payment in the native currency; there are lots of other ways to do it too and so another way a lot of BFT-type protocols handle rewards is the more amortized things. Thus, they'll go back and look at which node produced which fraction of the blocks over those 24 hours and then we'll just Dole out some amount of rewards proportionally to the number of blocks that different nodes created during that time.

Consequences:
This idea of block rewards it solves some problems but also creates some other ones. Let's look at the problems it solves:

1. **Incentivizes participation and block production.** One problem it would seem to solve is the problem we started with, which is why would nodes bother to run one of these consensus protocols, especially if there's a lot of work to be done in doing so and if you're paying people stuff that has real economic value you got to expect some people are going to find it in their interest to go through the trouble of running a node.

2. **Mechanism to grow supply of currency.** By creating new coins as rewards, the protocol ensures a continuous influx of cryptocurrency into circulation. Where do the coins come from? This goes all the way back to Bitcoin, this is yet another brilliant aspect of the Bitcoin protocol. Two different questions are solved in one Fell Swoop by block rewards. So first of all, obviously incentivizes block production, but then secondly that explains where the money comes from. Clockwork nodes are going to be producing these blocks every time a block is produced, new coins are going to be minted as time goes on the money supply will just grow and grow.

   A couple of comments specific to bitcoin while we're on the topic:
   First of all, you may know famously there's a hard cap on the number of Bitcoins. It will never exceed 21 million Bitcoins, that said, for the 13 plus years that have been in existence speaking here in late 2022, the supply of Bitcoins has been increasing steadily over that time, there have been steady block rewards that entire time. Eventually sufficiently far in the future, those block rewards are programmed to go to zero at which point the money supply will just will just stay fixed.

   A second interesting aspect about Bitcoin is that the protocol is unusual in that literally, the only way Bitcoins have ever been created is through block rewards. So, for every Bitcoin in existence, you can point to some block on the longest chain of the Bitcoin protocol and say this coin originated as part of the block reward of this specific block in the blockchain. So when the protocol was launched, there were zero Bitcoins in existence. Then the first block got created presumably by Nakamoto himself, running the protocol on their computer so that at the time that created 50 new Bitcoins then presumably again Nakamoto himself created the second block 10 minutes later, that was another 50 Bitcoins and so on.

Now these days while block rewards are still quite common and it's a way of growing the money supply. When most blockchain protocols launch, they also have a non-zero number of coins in existence so there's some kind of initial distribution of the Native currency to various stakeholders. Generally speaking, some coins go to the team that wrote the protocol, some coins go to investors, some coins go to the community, and so on. That's

kind of the most common thing you see; a mix of like an initial distribution of some non-zero but fixed amount of coins then plus sort of growth in the money supply over time through block rewards that distribute further coins to whoever is participating in the protocol.

10.2.2 Inflation rate

If you're wondering about the typical magnitude of these black rewards, one way to think about it is in terms of an inflation rate. You can look at what is the percentage increase in the money supply annually because these new coins are being minted with every block and the answer varies protocol to protocol, ranging from the low single digits to the high single digits. In many cases, initially in a protocol, you have a relatively high rate of inflation because you want to bootstrap a good set of nodes to start running the protocol and then you may see that the inflation rate drops slowly over time. For example in Bitcoin as discussed, when it was first launched, every block led to the minting of 50 new Bitcoins and in Bitcoin programmatically every four years or so, the black reward gets cut in half. So in 2012 it got cut from 50 to 25, and in 2016 got cut from 25 to 12.5, and then in 2020, it gotcut from 12.5 to 6.25 which is where it is now in late 2022 and then eventually, in a couple of years or even a little less that'll be cut to 3.8 Bitcoins.

Now we go through the problems cryptocurrencies create:

1. **Risk of newly incentivizing undesirable behavior :**

Whenever you introduce new incentives into a system and whatever your intuition may be about what those incentives are supposed to achieve, you need to take a step back and look at your new system and say "What are people actually incentivized to do now that I've introduced this new incentive system?". Block rewards are intended to motivate nodes to follow the blockchain protocol honestly, as soon as you get money for actions, you have to worry maybe they're even more incentivized to do something else. For example, it no longer makes sense to speak only about the kind of "honest nodes" who just obediently follow a consensus protocol and then Byzantine nodes that want to mess up the protocol. That's the dichotomy we've been using thus far and that's the traditional one you use in the analysis of consensus protocols, but once you have real money in the system and once you have block rewards, you still might have Byzantine nodes and there still might be people who want to take down your protocol. You might have few honest nodes for instance, maybe the team that launched the blockchain protocol in the first place is going to obediently follow the protocol that they wrote, but if it's a permissionless system where nodes can just join or not as they see fit even if they're not Byzantine you have to at least assume that they're profit maximizing and so if there's some strategy different from the one you had in mind and different from obediently following your protocol, if there's some different strategy that earns them more rewards, you somehow have to expect them to do it. You can think of this chapter as a famous case study that illustrates this exact point that when you introduce incentives, it can have unintended effects; maybe you incentivized to some extent the intended behavior you had in mind for nodes but maybe you incentivized even more other undesirable strategies.

Summary. In this chapter, the focus is on the relationship between specific details and broader lessons related to blockchain incentives and protocols. We discuss a particular unintended incentivized strategy known as "selfish mining," which relies on forking attacks and exploits the difficulty adjustment mechanism in Nakamoto consensus (used by Bitcoin). However, the key takeaway extends beyond this specific example. The broader lessons highlighted are:
1. Incentive Design Quandary: Blockchain protocol designers face a challenge in creating incentives to reward participants for their work while avoiding unintended behaviors. Careful consideration is required to introduce incentives effectively.
2. Applicability Across Protocols: The lesson of being cautious about introducing incentives applies to most permissionless blockchain protocols, not just Nakamoto consensus. The implications are relevant across various scenarios, even in life beyond blockchain.

10.3 Maximizing Block Rewards

Overview: In this section, we delve into the intricacies of Nakamoto consensus, with a specific focus on longest chain consensus and proof of work (PoW) civil resistance. We explore the concept of selfish mining and analyze how nodes' behavior may deviate from the protocol due to varying incentives. Throughout this section, we make key assumptions regarding the communication network and tie-breaking rules among competing longest chains. The ultimate goal is to shed light on the surprising result that honest behavior does not always constitute a Nash equilibrium in Nakamoto consensus protocols.

Setting of Nakamoto Consensus:

1. Focus on longest chain consensus and PoW civil resistance.
2. Assumes a fixed block reward, e.g., 6.25 Bitcoins per block in the Bitcoin protocol.
3. For each finalized block (sufficiently deep in the longest chain), the creator receives a fixed block reward.

Honest Behavior and Incentives

1. In a perfect world, nodes are incentivized to honestly follow longest chain consensus.
2. Create blocks that extend the current end of the longest chain and announce them immediately.
3. With PoW civil resistance, having an α fraction of the overall hash rate gives an α probability of creating the next block.

The question is that in Nakamoto consensus, what could go wrong?
The key Point is that it's true that in Nakamoto consensus if you have 10 of the overall hash rate and as usual making a random Oracle assumption and so on, you will be creating ballpark 10% of the blocks that get created but here's the thing: rewards are only doled out for blocks that get finalized, blocks that wind up on the longest chain(sufficiently deep in the longest chain). Blocks that get orphaned do not get a block reward so the worry then would be that some nodes might get their blocks orphaned at a higher rate than others.
If so, then we've got an issue because a node whose blocks are getting orphaned at a higher than average rate is going to be earning a less than expected fraction of the overall rewards.

To make this a little more concrete, let's page back to how difficult the adjustment Works in Nakamoto consensus where you have proof-of-work civil resistance. In proof of work, you have these difficulty parameters τ and the question is how should you set τ? What I've been saying is that generally, you set τ to target a given rate of block production. For instance in the Bitcoin protocol, the famous τ is tuned so that the average rate of black production is one block every 10 minutes. In fact, you're tuning it to target a particular rate of growth of the longest chain, for example in the blockchain protocol the difficulty parameter is adjusted periodically, meant to be roughly every two weeks or so now if one block is being added to the longest chain every 10 minutes, then over two weeks you expect there to be 2016 blocks added to the longest chain and every time the height of the longest chain grows by 2016, the Bitcoin protocol says how long did it take to produce these 2016 blocks by looking at the timestamps in the blocks and if it took a long time (more than two weeks) to produce those 2016 blocks on the longest chain, that means that the puzzles are too difficult. In order to make them easier, you increase the difficulty parameter τ. And if you created 2016 blocks faster than expected (meaning in less than two weeks), then the puzzle is still too easy so you decrease the difficulty threshold τ to make them harder.
But the point is that the difficulty parameters are being adjusted and constantly tuned so that 2016 blocks get added to the longest chain every two weeks. That does not speak about the number of blocks produced that might not be on the longest chain; maybe nothing was orphaned there were only 2016 blocks total during those two weeks and they will all end up in the longest chain or maybe 3016 blocks were produced during those two weeks a thousand of them were orphaned and then 2016 wound up on the longest chain from the perspective of difficulty adjustment. It does not matter, all that matters is the growth of the longest chain over two weeks.

Recall that block rewards are also doled out only to the blocks on the longest chain and not to the blocks that get orphaned. If the longest chain is growing at a predictable rate given that's what the difficulty adjustment algorithm is meant to do, for instance, 2016 blocks every two weeks, then the total amount of rewards being doled out is also growing at a steady rate so again for example in Bitcoin every two weeks you would expect 2016 times

six and a quarter Bitcoins so you know twelve thousand and change Bitcoins to be doled out in each two weeks.

Note:-

Orphaned Blocks and Reward Distribution

1. Rewards are only given to blocks on the longest chain.
2. Orphaned blocks (not on the longest chain) do not receive rewards.
3. The worry is that some nodes may have a higher rate of orphaned blocks, leading to lower rewards.

10.3.1 Proof of Work Difficulty Adjustment

In Proof of Work (PoW) based blockchains like Bitcoin, miners compete to solve complex cryptographic puzzles to add new blocks to the blockchain. The difficulty of these puzzles is adjusted periodically to ensure a steady and predictable block production rate. Here's how the difficulty adjustment works:

1. **Target Block Production Rate:** The blockchain protocol sets a target block production rate, which determines how frequently new blocks should be added to the blockchain. For example, in Bitcoin, the target block production rate is one block every 10 minutes on average.
2. **Difficulty Parameter Tau (τ):** The blockchain network uses a difficulty parameter, denoted as τ, to adjust the difficulty of the cryptographic puzzles miners must solve. The difficulty is inversely proportionate to τ. A higher value of τ corresponds to a lower difficulty, making it easier for miners to find a valid solution to the puzzle.
3. **Difficulty Adjustment Period:** The network regularly monitors the time taken to produce a certain number of blocks, typically referred to as the difficulty adjustment period. In Bitcoin, this period is roughly every two weeks.
4. **Calculating Difficulty Adjustment:** At the end of each difficulty adjustment period, the network examines the actual time taken to produce the target number of blocks and compares it to the expected time based on the target block production rate. If the time taken was longer than expected, the network concludes that the puzzles are too difficult. In this case, the difficulty parameter τ is reduced to make the puzzles easier for the next period. Conversely, if the time taken was shorter than expected, the network increases τ to make the puzzles harder for the next period.
5. **Ensuring Target Block Production Rate:** The difficulty adjustment aims to ensure that, on average, the target number of blocks is added to the blockchain within the specified time frame. For example, in Bitcoin, if one block is expected every 10 minutes, then over a two-week difficulty adjustment period, there should be approximately 2016 blocks (12 blocks per hour for 24 hours, multiplied by 14 days).
6. **Impact on Block Rewards:** Block rewards are only distributed to the blocks that get added to the longest chain. Orphaned blocks i.e. those that are not part of the longest chain, do not receive any rewards. As a result, the total rewards distributed in each difficulty adjustment period are proportional to the number of blocks on the longest chain during that period.

The difficulty adjustment mechanism ensures that the block production rate remains relatively stable, maintaining the security and consistency of the blockchain. However, as we further explore, this system can also be vulnerable to certain deviations known as "selfish mining," where nodes may deviate from honest behavior to maximize their rewards.

Selfish Mining and Deviating Strategy

In the context of Nakamoto consensus and PoW-based blockchain systems, selfish mining refers to a strategy where a node tries to maximize its block rewards by deviating from the expected honest behavior of following the longest chain consensus. In this strategy, nodes act as profit-maximizing entities rather than being purely obedient to the protocol.

The fundamental assumption behind selfish mining is that nodes aim to maximize their rewards by optimizing their block creation strategy. Instead of honestly propagating blocks that extend the current end of the

longest chain and competing with other miners for block creation, selfish miners look for opportunities to gain a strategic advantage and increase their share of the rewards.

Note:-

Selfish Mining and Deviating Strategy

1. Selfish mining refers to maximizing block rewards by deviating from honest behavior.
2. Nodes are treated as profit-maximizing entities, not just obediently following the protocol.

The key takeaway of this chapter (not necessarily intuitive or easy to see) is in fact yes. So, if you assume that all of the other nodes are just honestly following the longest chain protocol, there exists a deviation that you can make that boosts your share of the rewards beyond just your fraction of the hash rate. So you might have 10 of the hash rate but there is a sort of deviating strategy that will get you for example 12 of the blocks on the longest chain.

In this chapter, we will discuss deviations known as "selfish mining," which involves nodes maximizing their block rewards by not obediently following the protocol. Instead, nodes will act as profit-maximizing entities, considering block rewards of meaningful economic value.

The term "selfish mining" is self-explanatory, as it reflects the behavior of nodes prioritizing their gains over honest adherence to the consensus protocol. In the context of Nakamoto consensus and PoW-based systems, selfish mining refers to the production of blocks that optimize individual profits.
Nodes engaged in selfish mining are analogous to miners who put significant effort into finding valuable resources like gold. Similarly, in a proof-of-work system, nodes expend computational power to produce blocks, which we equate to mining in this context.
It is important to note that this chapter considers block rewards with meaningful economic value, as seen in systems like Bitcoin. As a result, nodes are motivated to act as profit-maximizing agents rather than mere followers of the protocol's guidelines. By studying selfish mining, we aim to understand how deviations from honest behavior can impact the stability and fairness of blockchain systems.

Key Takeaway: Obedience Not a Nash Equilibrium
The concept of a Nash equilibrium in game theory refers to an outcome where no participant has an incentive to unilaterally deviate from their chosen strategy, given the strategies chosen by all other participants. In other words, no player can improve their payoff by changing their strategy, assuming all other players' strategies remain unchanged.

In the context of Nakamoto consensus and selfish mining, the key takeaway is that obediently following Nakamoto consensus does not necessarily constitute a Nash equilibrium. In this scenario, nodes are assumed to honestly follow the longest chain consensus, creating blocks that extend the current end of the longest chain and announcing them immediately to others.
However, it has been shown that there exists a deviation strategy, known as selfish mining, that allows a node with a certain hash rate to earn more rewards than its expected fraction. This means that a node can improve its payoff by deviating from the honest behavior of following the longest chain consensus.
The selfish mining strategy involves maximizing block rewards by not strictly adhering to the protocol. Instead, nodes act as profit-maximizing entities, considering alternative strategies that can lead to higher rewards. This deviation from honest behavior creates an incentive for nodes to act selfishly and not follow the consensus protocol.
Since some nodes have an incentive to deviate from the protocol, the outcome of obediently following Nakamoto consensus does not qualify as a Nash equilibrium. At least one participant (a node) has an incentive to unilaterally deviate and improve its rewards, assuming that all other nodes continue to follow the consensus protocol.
This result may seem counterintuitive, as Nakamoto consensus was designed with incentives (block rewards) to encourage nodes to follow the protocol honestly. However, the presence of selfish mining strategies shows that there are scenarios where a node can gain more by not strictly adhering to the consensus rules.

It is important to note that this result does not imply a fatal flaw in Nakamoto consensus or blockchain protocols in general. It serves as a cautionary tale, demonstrating that even well-designed incentive mechanisms

can lead to unintended behaviors. Understanding these nuances is crucial for building robust and secure blockchain protocols in the real world.

Note:-

Key Takeaway: Obedience Not a Nash Equilibrium

1. Obediently following Nakamoto consensus does not necessarily constitute a Nash equilibrium.
2. There exists a deviation strategy (selfish mining) allowing a node with a certain hash rate to earn more rewards than its expected fraction.

10.3.2 Variants of Selfish Mining

Extreme Case: Node with More Than 50% Hash Rate

- A node with over 50% of the hash rate can use selfish mining to increase its share of rewards beyond its hash rate fraction.

When a node possesses more than 50% of the overall hash rate, it gains a significant advantage in the PoW consensus mechanism. It can secretly mine blocks in a selfish manner, withholding these blocks from the network while continuing to mine new blocks on its private chain. By doing so, the selfish node can create a longer private chain, surpassing the public chain created by other honest nodes. When it eventually reveals its private chain, it causes the honest nodes' blocks to be orphaned, and the selfish node receives higher rewards than its fraction of the total hash rate would suggest. This strategy allows the node to unfairly increase its share of the overall rewards.

Note:-

Assumptions

(i) perfect communication network("super-synchronous" / instant communication model)

(ii) honest nodes break ties arbitrarily (in effect, done by an adversary)

Honest Nodes' Tie-Breaking Assumption

1. Honest nodes break ties among multiple longest chains arbitrarily, allowing adversarial tie-breaking.
2. This assumption affects the analysis of selfish mining strategies.

In the context of Nakamoto consensus, there can be situations where multiple longest chains emerge due to conflicting blocks. Honest nodes need to decide which chain to extend, and in the absence of a clear rule, they may make arbitrary choices to break ties. However, this assumption can be exploited by a selfish node to its advantage. By manipulating the tie-breaking process, the selfish node can increase its chances of producing the longest chain, even if it has less than 50% of the total hash rate. Consequently, this affects the incentives for honest nodes and provides opportunities for the selfish node to deviate from following the longest chain consensus.

Relaxing Honest Nodes' Tie-Breaking Assumption

- Even with honest nodes breaking ties optimally, nodes with less than 50% hash rate can still be incentivized to deviate. The previous variant assumed that honest nodes may make suboptimal tie-breaking decisions, which could be exploited by a selfish node. However, even when honest nodes break ties optimally and strategically, there are still scenarios in which nodes with less than 50% hash rate are incentivized to deviate from the intended behavior of following the longest chain. This demonstrates that the incentive to engage in selfish mining is not solely dependent on the tie-breaking assumption but is a more fundamental characteristic of the PoW consensus mechanism.

Conclusion

1. Selfish mining can disrupt Nakamoto consensus, leading to nodes earning more rewards than expected.
2. This serves as a cautionary tale about the potential for unintended behaviors in complex incentive systems.
3. Although Nakamoto consensus has worked well in real-world applications like Bitcoin, understanding these issues is essential for designing robust blockchain protocols.

10.4 The Case of a 51% Miner

10.4.1 Warm up: A Profitable Deviation for the 51%

In this section, we're going to focus on Nakamoto Consensus, the underlying mechanism of blockchain protocols like Bitcoin. The key takeaway is that in Nakamoto Consensus, nodes are not generally incentivized to obediently follow the protocol, as they can earn higher profits by deviating from intended behavior.
We are going to start with the same assumptions we had in the previous section:

Assumptions

1. Instantaneous Communication: All honest nodes can communicate instantaneously by telepathy, making the negative results even stronger. In the context of Nakamoto Consensus, instantaneous communication ensures that nodes can quickly receive and propagate new blocks without any message delays. This facilitates a smooth growth of the blockchain without unnecessary forks or conflicts. This assumption doesn't bother us.

2. Adversarial Tie-Breaking: The same node contemplating a deviation can break ties among competing longest chains. The second assumption does bother us a lot. This assumption means that Node A, which seeks to deviate from honest behavior and orphan honest blocks, has control over how honest nodes resolve conflicts when there are competing chains. This adversarial tie-breaking allows Node A to consistently prioritize its blocks over those produced by honest nodes, ensuring that its alternative chain eventually becomes the longest chain.

Suppose: a node A controls 51% of the overall hash rate. Node A (with a strict majority of the overall hash rate) seeks to maximize its block rewards by deviating from the protocol and orphaning honest nodes' blocks.
Strategy 1: Honest Node Behavior
If Node A obeys the protocol and acts honestly, 100% of the hash rate follows the longest chain consensus. In the super-synchronous model (no message delays), forks are avoided, and the chain grows in an orderly fashion. Node A, with 51% hash rate, under our normal Oracle assumption, produces approximately 51% of the blocks on the longest chain. It is going to be producing roughly 51 of the overall blocks since every single block produced winds up in the longest chain in this thought experiment. That means node A will have contributed 51 of the blocks that wind up on the longest chain. For example, if we're talking about the Bitcoin protocol, then every two weeks we're expecting 2016 blocks, so $51 \times 2016 \times (6\frac{1}{4})$ Bitcoins; that's going to be the overall amount of block rewards that a node A earns.

Now the question is given that you know node A is under sort of no contract to do this, to follow the protocol obediently, is there some other strategy that node A could do instead under which it would reap a higher fraction of the block rewards? The answer is definitely yes!

Strategy 2: Deviation by Node A Node A aims to orphan all honestly produced blocks and create an alternative chain. Node A keeps trying to extend its chain to orphan honest blocks from the longest chain. As Node A has more than half the hash rate, it produces blocks more frequently than honest nodes, ensuring its chain eventually overtakes the longest chain.
Example of Strategy 2 (Orphaning Honest Blocks)

1. The scenario starts with a Genesis Block (B_0) (see more in Figure 10.1).

2. An honest node creates the first block, B_1, on the longest chain.

3. Node A's deviation begins here, as it attempts to orphan B_1. To do this, Node A tries to create its block, B'_1, extending from the Genesis Block (B_0), not B_1.

   - Node A aims to have its chain grow separately from the longest chain and keep extending B_0.

4. Suppose the next block, B_2, is honestly produced and added to the longest chain.

5. Node A continues to follow its deviation strategy by trying to create another block, B'_2, extending from B'_1, not from B_2.

   - Node A still aims to have its chain grow separately from the longest chain and extend B'_1 instead of B_2.

6. This process continues, with Node A trying to extend its alternative chain with blocks like B'_3, B'_4, and so on, while not following the longest chain.
    - At each step, Node A may succeed in creating new blocks that extend its alternative chain, but there is also a chance that honest nodes will create blocks on the longest chain.
    - If honest nodes create blocks on the longest chain, they will extend B_2 or B_3, ignoring Node A's B'_2 and B'_3.
7. Node A's strategy persists until, over time, it successfully produces more blocks on its alternative chain than honest nodes produce on the longest chain.
    - Since Node A controls more than half of the hash rate, it has a higher probability of producing blocks faster than honest nodes, increasing its chances of overtaking the longest chain.

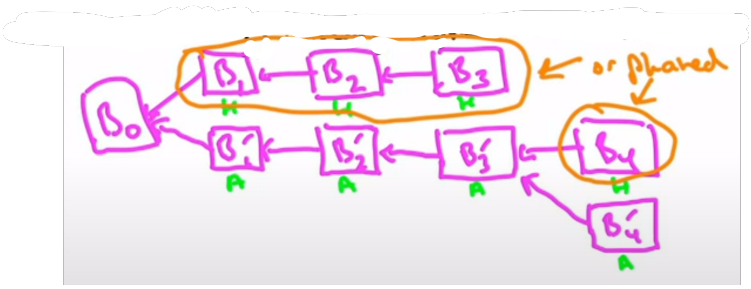


Figure 10.1: The example of strategy 2

Conclusion of Strategy 2: As Node A's alternative chain(A-blocks) grows, it will eventually catch up and overtake the longest chain(H-blocks) in terms of block height. This means that the longest chain will switch to Node A's chain, as it is no longer. Hence, Node A's chain becomes the new longest chain, and all block rewards will belong to Node A. When this happens, all honestly produced blocks (B_1, B_2, B_3, etc.) are orphaned, meaning they are no longer part of the longest chain. Instead, the longest chain now comprises only blocks produced by Node A.

Profit Maximization: Node A's strategy of deviating from honest behavior and orphaning honest blocks allows it to maximize its share of the block rewards. By having control over the majority of the hash rate, Node A can ensure its chain grows faster and replaces the longest chain.
The example above demonstrates how Node A, with more than 50% of the hash rate, can successfully deviate from the protocol to maximize its block rewards by orphaning honestly produced blocks. This strategy allows Node A to claim 100% of the block rewards, which is more profitable than following the protocol obediently.
Note that this strategy is specific to the extreme case where Node A controls more than half of the hash rate. In the next part, we will explore a variant where Node A has less than 50% hash rate and how the strategy might be tweaked to still boost Node A's share of the block rewards.

10.5 Selfish Mining (Worst-Case Tiebreaking

Now, we will explore the concept of deviating from the longest chain consensus protocol and its implications on node rewards. The key takeaway is that in general, deviating from this protocol can boost the rewards of a node. We will discuss examples and strategies to demonstrate this phenomenon.

10.5.1 Background

In the previous section, we examined a scenario where a large node with more than half of the hash rate deviated from the longest chain consensus protocol, leading to several issues. Now, we will focus on a current assumption of worst-case tie-breaking by honest nodes. Under this assumption, the size of the node, big or small, will not affect its profitability to deviate from the longest chain consensus. The deviations we will discuss are often referred to as "selfish mining," where nodes prioritize profit maximization over protocol obedience.

10.5.2 Selfish Mining and its History

Selfish mining is a concept that was introduced in the field of blockchain game theory. It was first formulated by Ayal and Sirira and was disseminated in 2013, followed by formal publication in 2014. At that time, Bitcoin was the dominant blockchain network, and Nakamoto consensus was the prevailing consensus protocol.

The term "selfish mining" refers to a behavior exhibited by certain nodes in the blockchain network. Instead of strictly adhering to the consensus protocol and validating transactions honestly, selfish mining nodes prioritize profit maximization. They strategically deviate from the standard longest chain consensus protocol to increase their share of block rewards.

Ayal and Sirira's work focused specifically on Nakamoto consensus, which is the underlying consensus mechanism used in Bitcoin. Under Nakamoto consensus, blocks are produced through a proof-of-work process, and nodes compete to find the solution to a cryptographic puzzle. The node that successfully solves the puzzle gets to create the next block and is rewarded with newly minted coins and transaction fees.

In their research, Ayal and Sirira explored various strategies that a selfish mining node could adopt to maximize its rewards. By exploiting certain weaknesses in the consensus protocol, a selfish mining node could potentially gain an advantage over other nodes and increase its share of block rewards.

It's worth noting that the concept of selfish mining was a significant development in the academic sphere. The research paper on selfish mining by Alan Sharir gained widespread attention and citations, making it one of the most referenced papers in the blockchain research community.

Over time, the term "selfish mining" has evolved beyond its original narrow meaning. While initially specific to Nakamoto consensus and profit-motivated deviations from honest behavior, it is now used in a broader sense. Today, "selfish mining" may refer to any behavior by block producers that deviates from the intended behavior of the protocol with the goal of increasing their revenue. This broader definition includes actions like increasing transaction fees or manipulating the application layer to maximize profits.

In conclusion, selfish mining, as introduced by Ayal and Sirira, is a significant concept in blockchain game theory, particularly in the context of Nakamoto consensus. It has sparked extensive research and discussions within the academic community, shedding light on the complexities of blockchain incentives and the behavior of rational actors within the network.

Now, we're going to discuss the main subject:
Let node A have $\alpha < \frac{1}{2}$ fraction of overall hash rate(All other nodes are honest).

10.5.3 Strategy: Orphaning Blocks

Let's delve into the strategy nodes use to increase their rewards. We'll consider a scenario where a node, denoted as Node A, has a hash rate α less than one-half. The two main ingredients of this strategy are as follows:
1. Orphaning Honestly Produced Blocks
Node A aims to have its blocks included in the longest chain while minimizing the number of honestly produced blocks on the chain. Unlike the previous section with 51% hash rate, Node A can't orphan all honestly produced blocks, but it still seeks to orphan the most recent ones.
2. Delaying Block Announcements
Node A takes advantage of the ability to delay block announcements. If Node A produces blocks that outpace the honestly produced blocks, it will only announce these blocks on a need-to-know basis, ensuring that honest nodes waste their efforts on the non-longest chain.

Example of the Strategy (Figure 10.2)

Let's illustrate the strategy with an example:
Step 1: We start with the Genesis block B_0.
Step 2: The next block is produced by an honest node, denoted as B_1.

Explanation: In the blockchain, new blocks are added to a chain, starting from the Genesis block. Block B_1 is the first block after the Genesis block, and it is honestly produced by one of the nodes following the protocol.
Step 3: Node A tries to extend the Genesis block to create an alternative block B'_1.
Explanation: Node A, with a hash rate α less than one-half, aims to increase its share of block rewards. It tries to produce an alternative block B'_1 that extends the Genesis block B_0. This is done to compete with block B_1 honestly produced by another node.
Step 4: If Node A successfully produces B'_1, B_1 gets orphaned.
Explanation: If Node A is successful in producing B'_1 and announces it to the network, the honest nodes will have a tie between two blocks, B_1 and B'_1, as botH-blocks extend the same predecessor (Genesis block). In this case, the adversarial tie-breaking mechanism allows Node A to extend B'_1, and B_1 becomes an orphaned block.
Step 5: The next block is produced by an honest node, denoted as B_2, extending B_1.
Explanation: The honest nodes keep following the protocol, and the next block B_2 is produced, extending B_1.
Step 6: Node A concedes B_1 and tries to extend B_1 to create B'_2.
Explanation: Node A concedes that it cannot orphan all honestly produced blocks. Instead, it focuses on the most recent ones. Here, Node A tries to extend block B_1 by producing B'_2. This is done to compete with block B_2 honestly produced by another node.
Step 7: If Node A successfully produces B'_2, B_2 gets orphaned.
Explanation: Similar to Step 4, if Node A successfully produces B'_2 and announces it, it creates a tie between B_2 and B'_2. As Node A has a lower hash rate, the adversarial tie-breaking mechanism favors the block B'_2 produced by Node A, leading to the orphaning of B_2.
Step 8: Node A keeps the existence of B'_3 private and lets honest nodes extend B_2.
Explanation: Node A continues the strategy of delaying block announcements. It produces a block B'_3 but does not immediately announce it to the network. Instead, it keeps it private. Meanwhile, the honest nodes continue extending block B_2.
Step 9: Once an honest node produces a block B_3 extending B_2, Node A announces B'_3.
Explanation: Node A waits until an honest node produces a block B_3 that extends B_2. At this point, Node A announces its private block B'_3. This is done strategically to trick the honest nodes into extending the non-longest chain.
Step 10: Honest nodes extend B'_3, orphaning B_3.
Explanation: The honest nodes, unaware of the existence of B'_3 due to Node A's delay in announcing it, keep extending block B'_3. As a result, B_3 becomes an orphaned block.

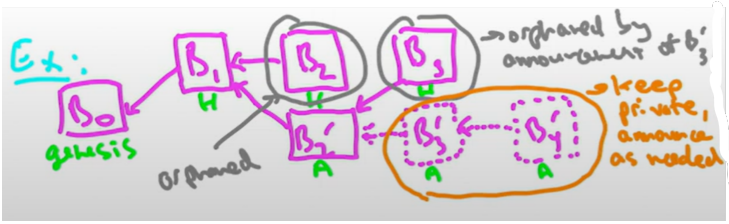


Figure 10.2: The example of strategy for the case of $\alpha < \frac{1}{2}$

Node A's strategy for increasing its share of block rewards involves two cases, depending on whether Node A is ahead in block production or if the honest nodes are ahead. Let h denote the maximum height of any block produced thus far. (Remember that by block height we just mean the number of steps since Genesis)

Case 1: Node A is Ahead

In this case, Node A is in an advantageous position, meaning it has already produced a block at the highest block height (denoted as h) among all the blocks that have been created. The strategy for Node A is straightforward:
Step 1: Node A extends its own block at height h to create an alternative block (denoted as B'_h).
Step 2: Node A delays announcing block B'_h, keeping it private for now.
Step 3: If Node A successfully produces another block at height h (denoted as $B"_h$), it also keeps this block private.
Step 4: Node A waits until an honest node produces a block at height h (denoted as B_h).

Step 5: Once an honest node announces B_h, Node A immediately announces its previously private block B'_h.
Step 6: Honest nodes, unaware of Node A's private block, start extending the longest chain with B_h as the new tip. Node A, however, already has its private block B'_h, so it extends its own chain.
Step 7: Honest nodes and Node A will now compete to extend their respective chains from heights h and h'. Node A aims to win this competition, which will result in orphaning B_h and keeping its own block B'_h as part of the longest chain.

Case 2: Honest Nodes are Ahead

In this case, the highest block height (denoted as h) among all the blocks produced so far was created by an honest node, not Node A. The strategy for Node A in this situation is as follows:
Step 1: Node A tries to produce a block at height h, extending the same block as the honest node (denoted as B'_h).
Step 2: Node A delays announcing block B'_h, keeping it private.
Step 3: If Node A successfully produces B'_h, it waits until an honest node creates a block at height h (denoted as B_h).
Step 4: Node A immediately announces its private block B'_h once an honest node announces B_h.
Step 5: Honest nodes will be unaware of Node A's private block and extend the longest chain from height h using B_h as the new tip. However, Node A's private block B'_h is already longer than any honestly produced block from height h.
Step 6: Honest nodes and Node A will compete to extend their respective chains from heights h and h'. Node A aims to win this competition, which will result in orphaning B_h and keeping its own block B'_h as part of the longest chain.

Throughout: Announce an A-block of height h as soon as there's an H-block at height h.

10.5.4 Analysis

Sure, here are the full notes in LaTeX form: The analysis focuses on a sequence of N consecutive rounds, where N is a reasonably large number, to study the fraction of blocks produced by Node A on the longest chain.

Step 1: Block Creation Expectations

Considering the N blocks, we expect an α fraction to be produced by Node A (A-Blocks) and a $(1-\alpha)$ fraction to be produced by honest nodes(H-blocks). This expectation relies on the random oracle assumption for block creation probability.

Step 2: All Node A-Blocks on Longest Chain

Every block created by Node A is guaranteed to be on the longest chain at the time of announcement. Thus, all Node A-blocks stay on the longest chain forever.

Step 3: Orphaning Honest Blocks

For every block created by Node A, an honestly produced block is orphaned due to adversarial tie-breaking. Thus, the number of orphaned honest blocks is the same as the number of Node A blocks.

Calculations:
Let's calculate the fraction of Node A's blocks on the longest chain compared to the total blocks.
# of A-blocks: The total number of Node A blocks is approximately αN, and all of them are on the longest chain.
# of H-blocks: The total number of honest blocks is approximately $(1-\alpha)N$, and approximately αN of them are orphaned, leaving $(1-2\alpha)N$ honest blocks on the longest chain.

Fraction of Node A Blocks: The fraction of Node A's blocks on the longest chain is:

$$\frac{\alpha N}{(1-\alpha)N} = \frac{\alpha}{1-\alpha}$$

For any positive value of α, the fraction $\frac{\alpha}{1-\alpha}$ is larger than α, meaning selfish mining boosts Node A's rewards, regardless of its hash rate. Even for small values of α, this strategy provides a significant increase in block rewards. The analysis also highlights a connection between selfish mining and the chain quality guarantees from Chapter 8. The strategy's impact will be discussed further in the upcoming sections.

10.6 Selfish Mining (Best-Case Tiebreaking)

In this section, we will examine the third and final version of the argument known as selfish mining, which involves deviating from the intended behavior and longest chain consensus. The focus will be on how this deviation can boost the share of block rewards earned by a node running the protocol.

10.6.1 Relaxing Assumptions

We will start by relaxing the assumption that we made in previous chapters, where we allowed the deviating node (node A) to control how honest nodes break ties among competing longest chains. This assumption was inherited from chapter eight, where we were studying the consistency and liveness properties of longest chain consensus. However, for analyzing selfish mining in a profit-maximizing node, we need to assume the opposite: the tie-breaking is not under the control of node A, and honest nodes will extend the longest chain with an honestly produced block at its tip.

Impact of Assumption:
Making this new assumption will make the analysis more challenging and complex than before. The strategy for the deviating node becomes more intricate, and the analysis becomes harder. The result will be weaker than before, indicating that deviating from longest chain consensus will be beneficial only for nodes possessing a substantial fraction of the hash rate (around 33% or more) but not for smaller nodes.

To illustrate the consequence of this assumption, we provide a simple example with a fork in the blockchain. Let's break down the explanation:

- **The Scenario:** There is a fork in the blockchain with two tips, B_1 and B_2, which have the same predecessor block B. B_1 is produced by an honest node (H-block), and B'_1 is produced by node A (A-block).
- **Previous Scenario:** In the previous section, the assumption was that node A had control over tie-breaking, so it could determine that all honest nodes would extend the A block B'_1. In that case, node A would be sure that its block B'_1 would end up in the longest chain.
- **Current Scenario:** However, in this video, the assumption is that honest nodes decide to extend the chain with an honestly produced block at its tip (H block) instead of the deviating node A block (A-block). Now, node A is not sure that its block B'_1 will end up in the longest chain.
- **Risk in Orphaning H-blocks:** Node A may try to orphan the H block B_1 by creating a competing block at the same height, which it successfully does (B'_1). However, for B_1 to be orphaned, node A needs to create a second block (B'_2) extending B'_1, and only then will honest nodes switch to extending the chain of A-blocks. Until this happens, honest nodes will keep extending the chain of H-blocks.
- **Consequences of Failure:** If an honest node creates a block (B_2) before node A produces B'_2, node A will be even further behind in chain length, and its attempts to orphan H-blocks will be increasingly challenging.
- **Complicated Analysis:** Due to these complex forces, the analysis becomes more intricate. The speaker mentions that the benefit of orphaning H-blocks by node A is uncertain and depends on the hash rate (Alpha) possessed by node A.
- **Impact on Deviation:** The uncertainty in the cost-benefit analysis indicates that whether deviating from the consensus protocol is profitable or not depends on the hash rate α. For sufficiently large nodes, deviating from the protocol may be a good idea, but for smaller nodes, it might not be beneficial.

Overall, the impact of this assumption on tie-breaking introduces new challenges and complexities in the analysis, making it harder to determine the profitability of the deviation for different hash rates possessed by the deviating node.

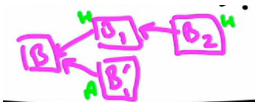


Figure 10.3: Consequence of the assumption

Modeling the Strategy: We will parameterize the analysis by the fraction of the hash rate possessed by the deviating node (node A), denoted as α. For $\alpha < \frac{1}{2}$, we will consider the case where all other nodes (1 - α fraction) follow the protocol honestly without profit maximization considerations. The goal is to understand if honesty is contagious in Nakamoto consensus or if a deviation leads to a Nash equilibrium.

10.6.2 Four Cases of Strategy

We will now describe the four cases of the strategy for node A based on different scenarios. We're going to keep track of two different block Heights h_p that's going to be the highest height the maximum height of every block that's been announced to everybody of course every honestly produced block has been announced to everybody because that's what honest nodes do node A will be selectively releasing announcements of blocks so all of the publicly announced blocks the highest height one among those is going to be h_p node A may also have some blocks that are secret that only it knows about and if there are any such blocks h_s is going to denote the highest height among all of them.
Remember that the height of a block in a blockchain is the number of hops that you need to get back to Genesis or equivalently it's the length of the chain that ends at that block. So the end of the longest chain is going to be exactly the block with the maximum height. For example, h_p is the maximum height of any publicly announced block from the perspective of the honest nodes who know only about the public announce blocks; they think that the longest chain is currently whichever one ends at the Block that has height h_p. Node A meanwhile may know about blocks at even higher Heights if node A ever chooses to release all those blocks that will create a new longest chain.

Case 1: No Secret Blocks ($h_p > h_s$)

In this case, Node A doesn't have any secret blocks, meaning all its blocks are publicly announced and known to everyone. Node A's goal is to maximize its share of the block rewards by getting as many of its blocks as possible into the longest chain, where block rewards are earned. The honest nodes, constituting the majority of the network (1 - α) fraction of the hash rate), are also following the protocol honestly and are trying to extend the longest chain, which is currently at height h_p.

Node A recognizes that its current position in the blockchain is not ahead of the honest nodes' longest chain. So, it decides to extend the publicly announced longest chain, trying to produce a new block, let's say B'_4, which would extend the current longest chain with a block of its own. In this way, Node A hopes to get its own block rewards and increase its share of the pie.

Meanwhile, the honest nodes continue extending the longest chain, aiming to produce an honestly produced block, say B_4, at height $h_p + 1$. This honest block will compete with Node A's secretly created block B'_4.

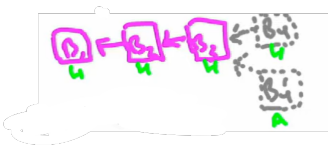


Figure 10.4: Case 1

Outcome 1: Node A's Success
With probability α, Node A succeeds in being the first one to produce a new block (B'_4) extending the longest chain. At this point, Node A keeps this newly created block secret and continues with its strategy.

Outcome 2: Honest Node's Success
With probability $1 - \alpha$, one of the honest nodes succeeds in producing an honestly generated block B_4 at height $h_p + 1$. Now, Node A's secretly created block B'_4 is not part of the longest chain, and the honest nodes continue extending their chain.

In either case, the situation remains in Case 1, and Node A continues trying to extend the longest chain, aiming to gain more control over the blockchain.

Case 2: Secret Block Created ($h_s = h_p + 1$)

In this case, Node A successfully creates a secret block B'_4, which extends the longest chain with a new block at height $h_p + 1$. Node A's privately held chain now has a height of $h_p + 1$, and it has a lead of one block over the publicly known longest chain maintained by the honest nodes.

Node A now faces a strategic decision: whether to continue keeping its newly created block B'_4 secret or to publicly announce it to the network. If it chooses to announce it, the honest nodes will realize that a new longest chain exists, starting from block B'_4. In this situation, the honest nodes will switch to extending the chain with Node A's blocks, as they always choose the longest chain with the latest honestly produced block at the tip.

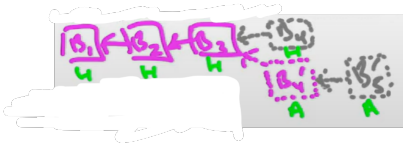


Figure 10.5: Case 2

Outcome 1: Node A Continues with Secret Block
With probability α, Node A decides to keep its newly created block B'_4 secret and tries to extend it further to increase its lead over the publicly known longest chain.
Outcome 2: Node A Announces Secret Block
With probability $1 - \alpha$, Node A chooses to announce its new block B'_4 to the network. This action changes the dynamics of the blockchain, leading to the network recognizing a new longest chain with Node A's blocks at its tip.

At this point, the situation moves to Case 4, where Node A has a secret block lead over the publicly known longest chain maintained by the honest nodes. It is important to note that Case 2 is reached only when Node A successfully creates a secret block that extends the longest chain, giving it a one-block lead. If it fails to do so, it will remain in Case 1 and continue extending the publicly known longest chain.

Case 3: Extending Private Chain($h_s = h_p$)

In Case 3, Node A is in a position where it already has a secret block B'_4 (B4 Prime) that it created back in Case 1. Node A is not willing to give up on this secret chain and is determined to extend it further. Meanwhile, the honest nodes are following their usual strategy of extending the end of the longest chain they know about.

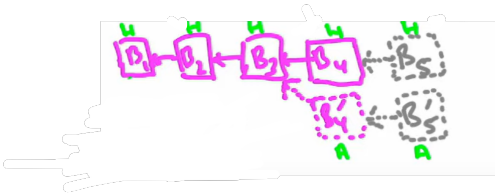


Figure 10.6: Case 3

Outcome 1: Node A's Success
If Node A successfully creates another secret block, say B'_5 (B5 Prime), before the honest nodes are able to

produce a block B_5, then Node A gains a lead of two blocks over the honest chain. In this situation, Node A announces both its secret blocks (B'_4 and B'_5).

By announcing both secret blocks, Node A effectively declares victory, making this new chain the universally recognized longest chain. The honest nodes were trying to create a block B_5, but they did not do it in time. Now, with the announcement of B'_4 and B'_5, these blocks are guaranteed membership in the longest chain. The honest nodes have no choice but to recognize this new chain and continue extending it.

This scenario is a win for Node A, as it managed to extend its secret chain and get its private blocks into the longest chain.

Outcome 2: Honest Node's Success
If the honest nodes successfully produce a block B_5 before Node A creates B'_5, then Node A's lead over the honest chain shrinks to just one block. In this case, Node A realizes that its chances of winning the race to extend the longest chain are diminishing. It decides to abandon its secret chain and return to Case 1, where it tries to extend the honest nodes' longest chain.

Node A accepts the fact that its secret block B'_4 got orphaned and it no longer has an advantage over the honest nodes. It acknowledges the honest nodes' new longest chain, which includes block B_5, and decides to move on.

Case 4: Large Secret Chain Lead ($h_s \geqslant h_p + 2$

In Case 4, Node A has managed to build a private chain with a lead of two or more blocks over the publicly known longest chain maintained by the honest nodes.

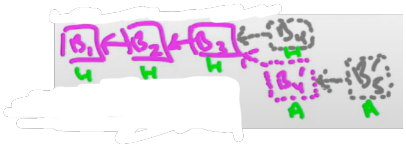


Figure 10.7: Case 4

Outcome 1: Node A Keeps Extending Secret Chain
If Node A continues to succeed in extending its secret chain, creating block B'_6 (B6 Prime), and maintains a lead of two or more blocks over the honest chain, it will keep extending its private chain.

Node A recognizes that its secret chain is significantly ahead of the honest chain, and as long as it can maintain this lead, it is in a favorable position to continue extending its private chain.

Outcome 2: Honest Node's Success
If the honest nodes manage to produce a block that closes the gap to just one block between Node A's secret chain and the honest chain, Node A faces a dilemma. It realizes that its advantage is diminishing rapidly, and there is a risk of its private blocks getting orphaned.

In this situation, Node A decides not to take any further risks and immediately announces all its secret blocks (e.g., B'_4, B'_5, B'_6) to the network. This action ensures that all of Node A's secret blocks become part of the longest chain and guarantees their inclusion.

By announcing all its secret blocks at once, Node A ensures that the deepest of its secret blocks (B'_6) becomes the new universally recognized longest chain. The honest nodes have no choice but to follow this new chain.

The announcement of multiple secret blocks might lead to the orphaning of some of the honestly produced blocks, but Node A accepts this trade-off to ensure its secret blocks are securely in the longest chain.

Outcome 3: Honest Node's Success (Lead Shrinks to One)
If the honest nodes manage to produce a block that further reduces the gap to just one block between Node A's secret chain and the honest chain, Node A takes immediate action to secure its secret blocks.

With a lead of only one block, Node A realizes that its position is becoming risky. It decides not to take any further chances and announces all its secret blocks (e.g., B'_4, B'_5, B'_6) to the network. This ensures the inclusion of all its secret blocks into the longest chain.

In this scenario, Node A announces three secret blocks at once, which guarantees their membership in the longest chain. These secret blocks orphan some of the honestly produced blocks, but Node A prioritizes securing its secret blocks and their inclusion in the longest chain.

Summary

The mining strategy employed by Node A is a complex interplay of different factors. It involves balancing the benefits of orphaning honestly produced blocks with the risk of having its secret blocks orphaned. Whether the strategy is a good idea or not depends on the hash rates of Node A and the honest nodes.

In Case 1, Node A follows the standard protocol and tries to extend the publicly known longest chain. It competes with the honest nodes to get its blocks into the chain and maximize its share of block rewards. However, this straightforward approach does not provide Node A with any special advantage.

In Case 2, Node A successfully creates a secret block that extends the longest chain, giving it a one-block lead. At this point, Node A faces a critical decision: whether to keep its newly created block secret or to announce it to the network. If Node A keeps it secret, it can continue extending its lead and possibly gain more control over the blockchain. If it announces the secret block, the honest nodes will recognize a new longest chain, starting from Node A's block. Node A's advantage, in this case, depends on whether it can continue creating secret blocks to maintain its lead.

Case 3 represents a riskier situation for Node A. Here, Node A already has one secret block, but the honest nodes have produced a block that narrows the gap to one block between Node A's secret chain and the honest chain. Node A attempts to create another secret block to maintain its lead, but if the honest nodes produce a block before it does, Node A will abandon its secret chain and go back to Case 1.

In Case 4, Node A enjoys a significant lead, with two or more secret blocks ahead of the publicly known chain. This is the most favorable situation for Node A, as it can keep extending its secret chain with lower risk. However, if the honest nodes manage to close the gap to one block, Node A will immediately announce all its secret blocks to ensure they get into the longest chain. This is a defensive move to avoid losing the advantage gained from having a large lead.

Analyzing the effectiveness of Node A's strategy requires non-trivial mathematical analysis. The outcome depends on the relative hash rates of Node A and the honest nodes. If Node A has a significant hash rate advantage, its strategy may lead to more of its blocks being included in the longest chain, potentially increasing its share of block rewards. On the other hand, if Node A's hash rate is relatively small compared to the honest nodes, its strategy may result in more of its blocks getting orphaned, leading to lower rewards.

Overall, the mining strategy employed by Node A demonstrates the complexities and challenges of deviating from the standard protocol in a blockchain network. It illustrates the importance of considering various factors, such as hash rates, block creation speed, and the risk of block orphaning, when devising a mining strategy. Mathematical analysis is necessary to determine the optimal strategy in different scenarios and to understand the potential impact on the overall network dynamics.

10.7 Markov Chain Analysis

Markov Chains are mathematical models used to describe random processes that exhibit the Markov property. The Markov property states that the future state of the system depends only on the current state and is independent of the past states. In other words, the future evolution of the system is memoryless and is determined solely by the current state.
In this section, we will analyze the strategy for node A in Nakamoto consensus. The strategy involves a tricky tug-of-war between two competing forces. The node may decide to deviate from obediently following the longest chain consensus, and the decision depends on the amount of hash rate that node A possesses. We will explore different cases and transitions using a Markov chain analysis.

10.7.1 Markov Chains and Transition Probabilities

Markov chains are useful in modeling random processes. In this context, we can represent the different states of the strategy with vertices and the probabilities of transitioning between states with directed edges.

A Markov chain has two main components: the set of states (vertices) and the transition probabilities (directed edges). We will use non-negative integers to label the states, representing how big a lead node A has in its secret chain relative to the publicly known chain.

10.8 States and Transitions

Let's define the states for our Markov chain:

- State 0: No lead at all (corresponds to case one).
  This state represents the situation where node A does not have any lead in its secret chain compared to the publicly known chain. In this case, node A simply tries to extend the end of the publicly known longest chain. There is no advantage for node A in this state.
- State 1: A lead of one (corresponds to case two).
  This state represents the scenario where node A has a lead of one in its secret chain over the publicly known chain. Node A has successfully extended the longest chain one block ahead. In this state, node A will keep the newly created block private and not reveal it to the network immediately.
- State f: Fork state (corresponds to case three).
  This state occurs when there is a tie between one of the blocks in node A's secret chain and the publicly known chain. Node A has succeeded in extending the longest chain, but an honest node has also produced a block at the same height. This results in a fork in the blockchain.
- State i: Lead of i (corresponds to case four, where $i \geqslant 2$).
  This state represents the situation where node A has a lead of i in its secret chain over the publicly known chain. Node A has successfully extended the longest chain i blocks ahead. In this state, node A will keep the i newly created blocks private.

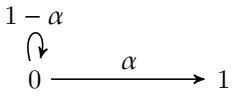
We will now explore the transitions between these states based on the strategy cases.

10.8.1 Strategy Cases and Transition Probabilities

Case One

Corresponds to State 0. Node A just tries to extend the end of the publicly known longest chain.

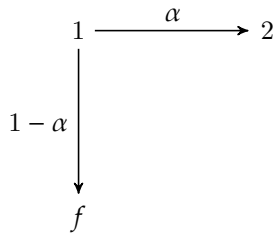
- If node A succeeds, it keeps the block private and transitions to State 1 with probability α.
- If node A fails (an honest node succeeds), it remains in State 0 with probability $1 - \alpha$. This means there is a $(1 - \alpha)$ chance that node A will still have no lead over the publicly known chain in the next step.



Case Two

Corresponds to State 1. Node A has a lead of one in its secret chain over the publicly known chain.

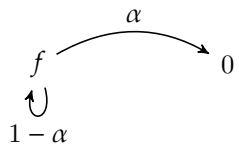
- If node A succeeds in extending the longest chain, it gains a lead of two and transitions to State 2 with probability α. This means there is an α chance that node A will have a lead of two in its secret chain over the publicly known chain in the next step.
- If node A fails (an honest node extends the longest chain), it transitions to State f (fork state) with probability $1 - \alpha$. This means there is a $(1 - \alpha)$ chance that node A will encounter a fork in the blockchain in the next step.



Case Three

Corresponds to State f (fork state). Node A has a tie between one of its private blocks and the publicly known block.

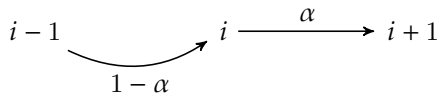
- If node A succeeds in extending its secret chain (breaking the tie), it transitions back to State 0 with probability α. This means there is an α chance that node A will restart its strategy with no lead over the publicly known chain.
- If node A fails (an honest node extends the longest chain), it also transitions back to State 0 with probability $1-\alpha$. This means there is a $(1-\alpha)$ chance that node A will restart its strategy with no lead over the publicly known chain.



Case Four

Corresponds to State i (lead of i). Node A has a lead of i in its secret chain over the publicly known chain.

- If node A succeeds in extending its secret chain, it increases its lead by one and transitions to State $i+1$ with probability α. This means there is an α chance that node A will have a lead of $(i+1)$ in its secret chain over the publicly known chain in the next step.
- If node A fails (an honest node extends the longest chain), it decreases its lead by one and transitions to State $i-1$ with probability $1-\alpha$. This means there is a $(1-\alpha)$ chance that node A will have a lead of $(i-1)$ in its secret chain over the publicly known chain in the next step.



Note: For $i = 2, 3, 4, \ldots$, the pattern continues with transitions between states $i-1$ and $i+1$.

Transition Probabilities:

1. From State 0 to State 1:
   - Probability of success (creating a block): α
   - Probability of failure (an honest node creates a block): $1-\alpha$
2. From State 1 to State 2:
   - Probability of success (creating a block): α
   - Probability of failure (an honest node creates a block): $1-\alpha$
3. From State 1 to State f(fork state):
   - Probability of success (creating a block): $1-\alpha$ (Node A loses the race)

- Probability of failure (an honest node creates a block): α (Node A wins the race)

4. From State f to State 0:
   - Probability of success (creating a block): $1-\alpha$ (Node A loses the race)
   - Probability of failure (an honest node creates a block): α (Node A wins the race)
5. From State f to State f:
   - Probability of success (creating a block): $1-\alpha$ (Node A loses the race)
   - Probability of failure (an honest node creates a block): α (Node A wins the race)
6. From State i to State $i+1$:
   - Probability of success (creating a block): $1-\alpha$ (Node A loses the race)
   - Probability of failure (an honest node creates a block): α (Node A wins the race)
7. From State i to State $i-1$:
   - Probability of success (creating a block): $1-\alpha$ (Node A loses the race)
   - Probability of failure (an honest node creates a block): α (Node A wins the race)

The following Figure 10.8 demonstrates the transition probability of all cases, all together:

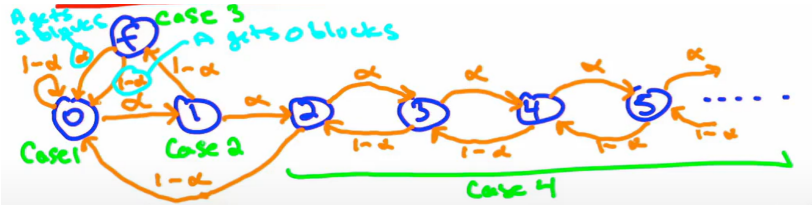


Figure 10.8: Transition Probabilities

Remember that α represents the probability that node A wins the race in creating the next block, and $1-\alpha$ represents the probability that an honest node wins the race in creating the next block.

These states and transition probabilities are essential in understanding how the proposed strategy behaves in different scenarios and help in analyzing its long-term implications in Nakamoto consensus. By analyzing the strategy using Markov chains, we can determine the probabilities of transitioning between different states and understand the potential outcomes for node A. This analysis helps us assess the effectiveness of the strategy and the block rewards that node A can expect in different scenarios.

10.8.2 Stationary Distributions

A stationary distribution in a Markov chain represents the equilibrium or long-run frequency of each state. In our case, it corresponds to the relative amount of time that the system spends in each state. The stationary probability of State i is denoted by π_i, which is also considered the long-run frequency of spending time in that state.

Computing Stationary Distribution:

To compute the stationary distribution for the specific Markov chain being discussed, let's consider the limit as a parameter T approaches infinity. The idea is to observe the system's behavior over an increasingly large number of transitions to understand the long-term probabilities of being in each state.

Let's delve deeper into the computation process:

The stationary probability of each state is denoted by π_i, where i represents the state in the Markov chain. For this specific Markov chain with two states (State 0 and State 1), we need to find the values of π_0 and π_1.

$$\pi_i = \lim_{T \to \infty} \frac{\text{Number of visits to State } i}{T}$$

This means that to find π_i, we need to consider the number of times the system visits State i during T transitions and take the limit of this ratio as T approaches infinity.

Note that the numerator of the ratio involves the count of consecutive transitions that land in State i (the number of visits to State i). The denominator T represents the total number of transitions or steps taken in the system.

The numerator, i.e., the number of visits to State i, is a random variable. It depends on how the transitions occur, and therefore, it may vary each time the experiment is conducted. However, although the numerator is a random variable, we can consider its expected value. This involves finding the average number of visits to State i over the long run.

The expectation of the number of visits to State i can be computed as:

$$\text{Expected visits to State } i = \lim_{T \to \infty} \frac{\text{Total visits to State } i}{T}$$

Now, we can rewrite the stationary probability π_i in terms of the expected visits to State i:

$$\pi_i = \lim_{T \to \infty} \frac{\text{Total visits to State } i}{T} = \lim_{T \to \infty} \frac{\text{Expected visits to State } i}{T}$$

As T approaches infinity, the denominator T becomes very large, making the fraction infinitesimally small. As a result, the stationary probability π_i becomes the limit of the expected number of visits to State i divided by an infinitely large number:

$$\pi_i = \lim_{T \to \infty} \frac{\text{Expected visits to State } i}{T}$$

It's important to note that the stationary distribution is independent of the initial state from which the system starts. This means that no matter where the system begins (State 0 or State 1), the long-run frequencies of being in each state will converge to the same values. There are closed-form formulas available for computing the stationary probabilities π_0 and π_1.

10.8.3 Strategy and Block Counting

We now explore a specific strategy involving the Markov chain described earlier. The strategy is designed to decide which block to extend in a blockchain system. The key objective is to understand how good this strategy is compared to the standard approach of following the longest chain.

Markov Chain and Strategy

The Markov chain in question consists of two states: State 0 and State 1. The transition probabilities between these states are denoted by α and $1 - \alpha$, respectively. The strategy is defined by different cases of block competition, which we refer to as cases one through four.

In case one, the strategy begins with an honest node extending the longest chain. On the other hand, in case two, node A creates a block but doesn't immediately add it to the longest chain. Instead, it keeps the block private, awaiting a possible lead in the future. If an honest node creates the next block in case two, the block created by node A in case one will be orphaned. Thus, it is too early to count this block toward node A's total.

Case three involves a more complex situation. Here, there is a competition between node A's private chain and the honest chain, which have the same height. Whichever chain gets extended first becomes part of the longest chain, and the other block becomes orphaned. In this case, node A is guaranteed to get two new blocks on the longest chain if it wins the competition. Likewise, if the honest nodes find the next block, they are also guaranteed to have two new blocks on the longest chain.

In case four, the situation is more straightforward. Every block created by node A is certain to end up on the longest chain, and every block created by the honest nodes will be orphaned. Therefore, transitioning from any state i to state $i + 1$ will always add one block to node A's total.

Block Counting Exercise

To evaluate the strategy's effectiveness, we conduct a block counting exercise for each transition in the Markov chain. This exercise helps determine how many blocks produced by node A and how many blocks produced by honest nodes end up on the longest chain.

Case One (State 0)
In case one, an honest node extends the longest chain. This block can be confidently counted as it will always be part of the longest chain.

Case Two (State 1)
In case two, node In case two, node A creates a block but doesn't immediately add it to the longest chain. Instead, it keeps the block private, waiting for the next block to be created. If an honest node creates the next block, the block created by node A will be orphaned. Thus, it is not yet appropriate to count this block toward node A's total.

Case Three (State f)
Case three involves a competition between node A's private chain and the honest chain. Whichever chain gets extended becomes part of the longest chain. If node A wins the competition, it is guaranteed to have two new blocks on the longest chain. Similarly, if the honest nodes win the competition, they are also guaranteed to have two new blocks on the longest chain.

Case Four (State 2, 3, 4, etc.)
In case four, every block created by node A is certain to end up on the longest chain. Conversely, every block created by the honest nodes will be orphaned. Therefore, every transition from state i to state $i+1$ adds one block to node A's total.

By conducting the block counting exercise for each transition, we identify the number of blocks that will be added to node A's total on the longest chain for each scenario. This analysis lays the foundation for evaluating the effectiveness of the strategy compared to the traditional approach of following the longest chain consensus.

10.8.4 Evaluating the Strategy

We can use the stationary distribution of the Markov chain to evaluate the strategy's performance. To do this, we need to compute the stationary probabilities π_i for each state. As stated, the stationary distribution represents the long-run frequency with which the system spends time in each state.

Expression for Honest Nodes' Blocks

The number of blocks produced by honest nodes that end up on the longest chain is given by:

$$\text{Honest nodes' blocks} = \pi_0 \cdot (1-\alpha) + \pi_f \cdot (1-\alpha) \cdot 2$$

The term $\pi_0 \cdot (1-\alpha)$ represents the probability that an honest node creates a block in State 0 (case one) and adds it to the longest chain. This happens with a probability of $(1-\alpha)$ when the system is in State 0 (case one), and π_0 is the stationary probability of being in State 0. The term $\pi_f \cdot (1-\alpha) \cdot 2$ accounts for the probability that an honest node extends the longest chain when the system is in State F (case three). In this case, two new blocks are guaranteed to be added to the longest chain (one for the publicly announced block and one for the private block of node A). π_f is the stationary probability of being in State F, and $(1-\alpha)$ is the probability that an honest node finds the next block.

Expression for Node A's Blocks

The number of blocks produced by node A that end up on the longest chain is given by:

$$\text{Node A's blocks} = \sum_{i=1}^{\infty} \pi_i \cdot \alpha + \pi_f \cdot \alpha \cdot 2$$

The term $\sum_{i=1}^{\infty} \pi_i \cdot \alpha$ represents the probability that node A creates a block in State i (case two) and successfully extends its private chain to State $i+1$. This is the case for all states from 1 to infinity. The summation considers

all possible states in which node A produces a block, weighted by their stationary probabilities π_i. The term $\pi_f \cdot \alpha \cdot 2$ accounts for the probability that node A extends the public chain from State F (case three) and adds its private block and the publicly announced block to the longest chain. The stationary probability π_f is used, and α represents the probability of node A finding the next block.

By evaluating the expressions for the number of blocks produced by node A and honest nodes, we can determine how good the strategy is compared to following the standard longest chain consensus. By comparing the results, we can identify the specific scenarios where this strategy outperforms the traditional approach.

10.8.5 Final Calculations

The main question is: What is the fraction of blocks on the longest chain that node A will have if it follows this strategy? We compare this fraction with the standard approach of honestly following the longest chain consensus, where node A would get a share of the blocks proportional to its hash rate.

Computing the Stationary Distribution

Linear Relationships in the Stationary Distribution
To compute the stationary distribution, we identify a few simple linear relationships between the stationary probabilities. These relationships are derived from the structure of the Markov chain.

- **Condition 1:** $\pi_1 = \alpha \cdot \pi_0$

  This condition represents the relationship between π_0 and π_1. Here, π_0 is the probability of the system being in State 0, which corresponds to the state where node A has an empty private chain and the public longest chain is one block ahead. π_1 is the probability of the system being in State 1, where node A has successfully mined the next block on its private chain and has caught up to the public longest chain.

  The equation $\pi_1 = \alpha \cdot \pi_0$ indicates that the probability of transitioning from State 0 to State 1 is given by α, which is the probability that node A successfully finds the next block. In other words, if node A has an empty private chain (State 0), the chance of it mining the next block and reaching State 1 is α. This makes intuitive sense as α represents node A's hash rate relative to the total hash rate of the network.

- **Condition 2:** $\pi_f = (1 - \alpha) \cdot \pi_1$

  Condition 2 establishes a relationship between π_1 and π_f. In this context, π_f represents the probability of the system being in State f, which denotes the state where the public longest chain is F blocks ahead of node A's private chain. Here, f can be any positive integer.

  The equation $\pi_f = (1 - \alpha) \cdot \pi_1$ means that the probability of transitioning from State 1 to State F is given by $(1 - \alpha)$. In other words, if node A has successfully mined the next block on its private chain (State 1), the chance of the public longest chain being f blocks ahead (State f) is $(1-\alpha)$. This reflects the probability that an honest node mines the next block and extends the public longest chain while node A is working on its private chain.

- **Condition 3:** $\pi_i \cdot \alpha = \pi_{i+1} \cdot (1 - \alpha)$ for all positive integers i

  Condition 3 provides a general relationship between π_i and π_{i+1} for all positive integers i. Here, π_i represents the probability of the system being in State i, and π_{i+1} represents the probability of the system being in the next state, State $i + 1$.

  The equation $\pi_i \cdot \alpha = \pi_{i+1} \cdot (1 - \alpha)$ states that the probability of transitioning from State i to State $i + 1$ is given by α, while the probability of transitioning from State $i + 1$ back to State i is given by $(1 - \alpha)$. This captures the dynamics of the Markov chain, where at each step, node A either mines a block and advances its private chain with probability α, or an honest node mines a block and extends the public longest chain with probability $(1 - \alpha)$.

By combining these linear relationships, we can see how all the stationary probabilities π_i can be expressed in terms of π_0. The sum of all stationary probabilities should be equal to 1, which allows us to solve for π_0 explicitly. Once π_0 is determined, all other stationary probabilities can be obtained using the linear relationships.

10.8.6 Ratio of Block Rewards

Once we have the stationary probabilities, we can compute the ratio of block rewards obtained by node A when following the selfish mining strategy to the rewards it would get by following the longest chain consensus.

The final ratio is a function of α, representing the fraction of blocks that node A successfully mines, and is given by:

$$\text{Ratio} = \frac{4\alpha^3 - 9\alpha^2 + 4\alpha}{\alpha^3 - 2\alpha^2 - \alpha + 1}$$

This ratio serves as a measure of how much more block rewards node A can gain through the selfish mining strategy compared to honest mining. If the ratio is greater than 1, it indicates that selfish mining is more profitable for node A, while a ratio less than 1 means that honest mining is more advantageous.

The analysis of selfish mining shows that if node A has a hash rate greater than one-third of the network's total hash rate, it is incentivized to deviate from the honest longest chain consensus. By employing the selfish mining strategy, node A can obtain a larger share of block rewards than it would get by following the standard approach.

While the specific strategy analyzed in this section is not optimal, it is still close to the optimal strategy. There are more sophisticated strategies that can provide even better rewards when node A's hash rate is slightly below one-third. However, when node A's hash rate is less than one-third, it is in its best interest to follow the longest chain consensus honestly.

Selfish mining analysis provides valuable insights into the incentives and behaviors of miners in a blockchain network and has significant implications for the security and stability of the network.

10.9 Discussion

In this final part of chapter 10, we will review what we have learned from the analysis of selfish mining, deviations from honest behavior, and Nakamoto consensus. We will also discuss the practical implications of this analysis.

Selfish Mining and Nash Equilibrium In this chapter, we started by emphasizing the crucial aspect of introducing a native cryptocurrency into a blockchain. It enables nodes to receive rewards for creating blocks. However, it is not always the case that incentivizing nodes to follow the longest chain consensus protocol results in a Nash equilibrium, where all nodes behave honestly.

Three different scenarios were presented to illustrate how selfish mining strategies can provide an incentive for nodes to deviate from honest behavior. The first scenario involves a node with 51% of the hash rate, which can guarantee 100% of the block rewards by consistently orphaning honestly produced blocks. The second scenario considers nodes with less than 50% of the hash rate but with the ability to influence how honest nodes resolve competing longest chains. In this case, a slightly more complicated strategy boosts the deviating node's share of the block rewards regardless of its hash rate. The third scenario is the most complex, where a node neither has 51% of the hash rate nor control over communication networks. Despite this, it can still benefit from selfish mining deviations, but only if it possesses more than a third of the total hash rate.

Academic Literature Influence This chapter also acknowledges the influential paper by Ayal and Sirer, published in 2014, which has sparked a vast array of follow-up works. These subsequent studies explore various aspects of selfish mining and deviations in different blockchain protocols, such as Ethereum 1.0 and various proof-of-stake designs. The academic community's exploration into these areas has been instrumental in understanding the implications of these strategies.

Practice and Implications Despite the theoretical analysis suggesting potential deviations, there is no evidence of nodes using these strategies in the 13 years of Bitcoin's existence.

Possible explanations for this lack of observed deviations include:

1. Capital Requirements: It is challenging to accumulate enough hash rate to execute the selfish mining strategy effectively, which might require a substantial investment in specialized hardware.

2. Delayed Payoff: The benefits of the deviation would take a long time to materialize due to the difficulty adjustment mechanism in Nakamoto consensus. This delayed payoff increases the risk and reduces the incentive for potential attackers.

3. Potential Price Crash: Some argue that public knowledge of an attack might lead to a drop in the price of the native cryptocurrency, resulting in the attacker losing wealth. However, this argument is not entirely convincing since prices are not always as sensitive to such attacks as believed.

Despite not being observed in practice, we should underscore the importance of the game-theoretic analysis in blockchain protocols. It demonstrates that in the context of cryptocurrencies and native incentives, the behavior of nodes cannot be solely classified as honest or Byzantine, and a more nuanced analysis is required.

Conclusion
Understanding game theory and incentive structures in blockchain protocols is crucial. The analysis presented in this chapter emphasizes the need for a nuanced approach to node behavior, beyond simple classifications of honest or Byzantine. While selfish mining strategies may not have been observed in practice so far, it is essential to remain vigilant and thoroughly assess the incentive mechanisms within blockchain systems. Future considerations should be given to revenue streams, including block rewards, transaction fees, and value from transactions. The upcoming chapter on Ethereum's EIP 1559 will delve further into transaction fee mechanism design. Stay tuned!

A creates a block but doesn't immediately add it to the longest chain. Instead, it keeps the block private, waiting for the next block to be created. If an honest node creates the next block, the block created by node A will be orphaned. Thus, it is not yet appropriate to count this block toward node A's total.

Case Three (State f)
Case three involves a competition between node A's private chain and the honest chain. Whichever chain gets extended becomes part of the longest chain. If node A wins the competition, it is guaranteed to have two new blocks on the longest chain. Similarly, if the honest nodes win the competition, they are also guaranteed to have two new blocks on the longest chain.

Case Four (State 2, 3, 4, etc.)
In case four, every block created by node A is certain to end up on the longest chain. Conversely, every block created by the honest nodes will be orphaned. Therefore, every transition from state i to state $i + 1$ adds one block to node A's total.

By conducting the block counting exercise for each transition, we identify the number of blocks that will be added to node A's total on the longest chain for each scenario. This analysis lays the foundation for evaluating the effectiveness of the strategy compared to the traditional approach of following the longest chain consensus.

10.9.1 Evaluating the Strategy

We can use the stationary distribution of the Markov chain to evaluate the strategy's performance. To do this, we need to compute the stationary probabilities π_i for each state. As stated, the stationary distribution represents the long-run frequency with which the system spends time in each state.

Expression for Honest Nodes' Blocks

The number of blocks produced by honest nodes that end up on the longest chain is given by:

$$\text{Honest nodes' blocks} = \pi_0 \cdot (1 - \alpha) + \pi_f \cdot (1 - \alpha) \cdot 2$$

The term $\pi_0 \cdot (1 - \alpha)$ represents the probability that an honest node creates a block in State 0 (case one) and adds it to the longest chain. This happens with a probability of $(1 - \alpha)$ when the system is in State 0 (case one), and π_0 is the stationary probability of being in State 0. The term $\pi_f \cdot (1 - \alpha) \cdot 2$ accounts for the probability that an honest node extends the longest chain when the system is in State F (case three). In this case, two new blocks are guaranteed to be added to the longest chain (one for the publicly announced block and one for the private block of node A). π_f is the stationary probability of being in State F, and $(1 - \alpha)$ is the probability that an honest node finds the next block.

Expression for Node A's Blocks

The number of blocks produced by node A that end up on the longest chain is given by:

$$\text{Node A's blocks} = \sum_{i=1}^{\infty} \pi_i \cdot \alpha + \pi_f \cdot \alpha \cdot 2$$

The term $\sum_{i=1}^{\infty} \pi_i \cdot \alpha$ represents the probability that node A creates a block in State i (case two) and successfully extends its private chain to State $i+1$. This is the case for all states from 1 to infinity. The summation considers all possible states in which node A produces a block, weighted by their stationary probabilities π_i. The term $\pi_f \cdot \alpha \cdot 2$ accounts for the probability that node A extends the public chain from State F (case three) and adds its private block and the publicly announced block to the longest chain. The stationary probability π_f is used, and α represents the probability of node A finding the next block.

By evaluating the expressions for the number of blocks produced by node A and honest nodes, we can determine how good the strategy is compared to following the standard longest chain consensus. By comparing the results, we can identify the specific scenarios where this strategy outperforms the traditional approach.

10.9.2 Final Calculations

The main question is: What is the fraction of blocks on the longest chain that node A will have if it follows this strategy? We compare this fraction with the standard approach of honestly following the longest chain consensus, where node A would get a share of the blocks proportional to its hash rate.

Computing the Stationary Distribution

Linear Relationships in the Stationary Distribution
To compute the stationary distribution, we identify a few simple linear relationships between the stationary probabilities. These relationships are derived from the structure of the Markov chain.

- **Condition 1:** $\pi_1 = \alpha \cdot \pi_0$

  This condition represents the relationship between π_0 and π_1. Here, π_0 is the probability of the system being in State 0, which corresponds to the state where node A has an empty private chain and the public longest chain is one block ahead. π_1 is the probability of the system being in State 1, where node A has successfully mined the next block on its private chain and has caught up to the public longest chain.

  The equation $\pi_1 = \alpha \cdot \pi_0$ indicates that the probability of transitioning from State 0 to State 1 is given by α, which is the probability that node A successfully finds the next block. In other words, if node A has an empty private chain (State 0), the chance of it mining the next block and reaching State 1 is α. This makes intuitive sense as α represents node A's hash rate relative to the total hash rate of the network.

- **Condition 2:** $\pi_f = (1-\alpha) \cdot \pi_1$

  Condition 2 establishes a relationship between π_1 and π_f. In this context, π_f represents the probability of the system being in State f, which denotes the state where the public longest chain is F blocks ahead of node A's private chain. Here, f can be any positive integer.

  The equation $\pi_f = (1-\alpha) \cdot \pi_1$ means that the probability of transitioning from State 1 to State F is given by $(1-\alpha)$. In other words, if node A has successfully mined the next block on its private chain (State 1), the chance of the public longest chain being f blocks ahead (State f) is $(1-\alpha)$. This reflects the probability that an honest node mines the next block and extends the public longest chain while node A is working on its private chain.

- **Condition 3:** $\pi_i \cdot \alpha = \pi_{i+1} \cdot (1-\alpha)$ for all positive integers i

  Condition 3 provides a general relationship between π_i and π_{i+1} for all positive integers i. Here, π_i represents the probability of the system being in State i, and π_{i+1} represents the probability of the system being in the next state, State $i+1$.

  The equation $\pi_i \cdot \alpha = \pi_{i+1} \cdot (1-\alpha)$ states that the probability of transitioning from State i to State $i+1$ is given by α, while the probability of transitioning from State $i+1$ back to State i is given by $(1-\alpha)$. This

captures the dynamics of the Markov chain, where at each step, node A either mines a block and advances its private chain with probability α, or an honest node mines a block and extends the public longest chain with probability $(1 - \alpha)$.

By combining these linear relationships, we can see how all the stationary probabilities π_i can be expressed in terms of π_0. The sum of all stationary probabilities should be equal to 1, which allows us to solve for π_0 explicitly. Once π_0 is determined, all other stationary probabilities can be obtained using the linear relationships.

10.9.3 Ratio of Block Rewards

Once we have the stationary probabilities, we can compute the ratio of block rewards obtained by node A when following the selfish mining strategy to the rewards it would get by following the longest chain consensus.

The final ratio is a function of α, representing the fraction of blocks that node A successfully mines, and is given by:

$$\text{Ratio} = \frac{4\alpha^3 - 9\alpha^2 + 4\alpha}{\alpha^3 - 2\alpha^2 - \alpha + 1}$$

This ratio serves as a measure of how much more block rewards node A can gain through the selfish mining strategy compared to honest mining. If the ratio is greater than 1, it indicates that selfish mining is more profitable for node A, while a ratio less than 1 means that honest mining is more advantageous.

The analysis of selfish mining shows that if node A has a hash rate greater than one-third of the network's total hash rate, it is incentivized to deviate from the honest longest chain consensus. By employing the selfish mining strategy, node A can obtain a larger share of block rewards than it would get by following the standard approach.

While the specific strategy analyzed in this section is not optimal, it is still close to the optimal strategy. There are more sophisticated strategies that can provide even better rewards when node A's hash rate is slightly below one-third. However, when node A's hash rate is less than one-third, it is in its best interest to follow the longest chain consensus honestly.

Selfish mining analysis provides valuable insights into the incentives and behaviors of miners in a blockchain network and has significant implications for the security and stability of the network.

10.10 Discussion

In this final part of chapter 10, we will review what we have learned from the analysis of selfish mining, deviations from honest behavior, and Nakamoto consensus. We will also discuss the practical implications of this analysis.

Selfish Mining and Nash Equilibrium In this chapter we started by emphasizing the crucial aspect of introducing a native cryptocurrency into a blockchain. It enables nodes to receive rewards for creating blocks. However, it is not always the case that incentivizing nodes to follow the longest chain consensus protocol results in a Nash equilibrium, where all nodes behave honestly.

Three different scenarios were presented to illustrate how selfish mining strategies can provide an incentive for nodes to deviate from honest behavior. The first scenario involves a node with 51% of the hash rate, which can guarantee 100% of the block rewards by consistently orphaning honestly produced blocks. The second scenario considers nodes with less than 50% of the hash rate, but with the ability to influence how honest nodes resolve competing longest chains. In this case, a slightly more complicated strategy boosts the deviating node's share of the block rewards regardless of its hash rate. The third scenario is the most complex, where a node neither has 51% of the hash rate nor control over communication networks. Despite this, it can still benefit from selfish mining deviations, but only if it possesses more than a third of the total hash rate.

Academic Literature Influence This chapter also acknowledges the influential paper by Ayal and Sirer, published in 2014, which has sparked a vast array of follow-up works. These subsequent studies explore various

aspects of selfish mining and deviations in different blockchain protocols, such as Ethereum 1.0 and various proof-of-stake designs. The academic community's exploration into these areas has been instrumental in understanding the implications of these strategies.

Practice and Implications Despite the theoretical analysis suggesting potential deviations, there is no evidence of nodes using these strategies in the 13 years of Bitcoin's existence.

Possible explanations for this lack of observed deviations include:

1. Capital Requirements: It is challenging to accumulate enough hash rate to execute the selfish mining strategy effectively, which might require a substantial investment in specialized hardware.

2. Delayed Payoff: The benefits of the deviation would take a long time to materialize due to the difficulty adjustment mechanism in Nakamoto consensus. This delayed payoff increases the risk and reduces the incentive for potential attackers.

3. Potential Price Crash: Some argue that public knowledge of an attack might lead to a drop in the price of the native cryptocurrency, resulting in the attacker losing wealth. However, this argument is not entirely convincing since prices are not always as sensitive to such attacks as believed.

Despite not being observed in practice, we should underscore the importance of the game-theoretic analysis in blockchain protocols. It demonstrates that in the context of cryptocurrencies and native incentives, the behavior of nodes cannot be solely classified as honest or Byzantine, and a more nuanced analysis is required.

Conclusion

Understanding game theory and incentive structures in blockchain protocols is crucial. The analysis presented in this chapter emphasizes the need for a nuanced approach to node behavior, beyond simple classifications of honest or Byzantine. While selfish mining strategies may not have been observed in practice so far, it is essential to remain vigilant and thoroughly assess the incentive mechanisms within blockchain systems. Future considerations should be given to revenue streams, including block rewards, transaction fees, and value from transactions. The upcoming chapter on Ethereum's EIP 1559 will delve further into transaction fee mechanism design. Stay tuned!

Chapter 11

Transaction Fees and EIP-1559

In this chapter, we're focusing on different economic issues related to blockchain protocols that have a native cryptocurrency.

11.1 Recap

In the previous chapter (chapter 10), we discussed five reasons for caring about cryptocurrencies in the context of blockchain protocols. First, cryptocurrencies are interesting in their own right. Second, having a native currency makes blockchain protocols more versatile and functional. Third, it enables convenient reward mechanisms for contributors to the protocol. Fourth, native currencies can incentivize desired behavior and discourage unwanted actions. Fifth, cryptocurrencies are crucial for proof-of-stake, a more energy-efficient alternative to proof-of-work consensus.
This chapter focuses on using a native cryptocurrency to charge for usage, specifically through transaction fees. Transaction fees play a vital role in blockchains, given the finite processing power and block size of these networks.

11.2 Transaction Fees and Economic Efficiency

11.2.1 Blockchains and Limited Throughput

Blockchains, such as Bitcoin and Ethereum, face a fundamental challenge when it comes to transaction processing - limited throughput. This limitation arises due to two main factors: the rate of block production and the block size.

In Bitcoin, blocks are produced at an average rate of one block every 10 minutes. Similarly, in Ethereum, new blocks are created approximately every 13 seconds. These time intervals determine how frequently new transactions can be added to the blockchain. While the rate of block production can be adjusted through protocol updates, there are practical constraints to consider.

Another critical factor affecting the throughput is the block size. In the past, Bitcoin had a hard limit of one megabyte per block, leading to debates within the community about increasing the block size to accommodate more transactions. Ethereum, on the other hand, measures block size using a concept called "gas," which represents the computational and storage costs of processing a transaction. Each block in Ethereum has a cap on the total gas that can be used, which effectively limits the number of transactions that can fit within a block.

These limitations mean that both Bitcoin and Ethereum, like many other blockchain networks, can only process a relatively small number of transactions per unit of time. For example, Bitcoin's throughput is far less than 10 transactions per second, and Ethereum's throughput is less than 20 transactions per second. In comparison, traditional payment networks like Visa can process thousands of transactions per second.

Challenges in Scaling Blockchains: The restricted throughput in blockchains raises concerns about scalability and widespread adoption. Simply increasing the block size or rate of block production is not a viable solution due to the goal of permissionless consensus and low barriers to entry. All nodes in the network need to be able to process transactions, and this requirement places a limit on the number of transactions per second. To address

these challenges, developers and researchers have been working on Layer Two scaling solutions, which aim to enable a significant increase in the number of transactions that can be processed off-chain while still securing the network using the underlying blockchain.

Despite the current limitations, it is crucial to maintain a balance between scalability and the core principles of blockchain networks, such as permissionless consensus and low barriers to entry. These principles ensure that anyone can participate in the network and that all nodes contribute to transaction processing. The limited throughput of blockchains like Bitcoin and Ethereum presents a significant challenge to achieving mainstream adoption and scalability. As the demand for blockchain-based transactions continues to grow, addressing these limitations will be critical to realizing the full potential of blockchain technology in various industries.

Competition for Block Space: Blockchains like Bitcoin and Ethereum have a limited capacity to process transactions due to the rate of block production and the size of each block. For instance, in Bitcoin, blocks are produced approximately every 10 minutes, while in Ethereum, they are produced every 13 seconds. Additionally, both blockchains have a finite cap on the number of transactions that can fit into a block. For example, Ethereum has a hard limit of 15 million gas per block, which is a proxy for computational and storage costs.

The limited throughput of these blockchains presents a challenge for handling a large number of transactions. As a result, there is fierce competition among transactions to be included in the blocks. Miners in the case of Bitcoin or validators in the case of Ethereum must decide which transactions to include and which to exclude from each block.

Since block space is a scarce resource, it becomes essential to prioritize valuable transactions that bring significant value to their creators. On the other hand, transactions that may be considered spam or have frivolous uses should be excluded to ensure efficient usage of the blockchain.

One may wonder how to determine the value of a transaction and whether it should be included in a block. Asking the transaction creators directly may not be effective, as they could overstate the value of their transactions to get them included.

The solution to this challenge lies in the use of transaction fees. By charging a non-trivial transaction fee for executing a transaction on the blockchain, the protocol incentivizes users to indicate the value they place on their transactions. Transactions that are genuinely valuable to their creators will be more willing to pay higher transaction fees, while those with less value will be discouraged from spending large fees. In this way, transaction fees act as a screening mechanism, helping prioritize high-value transactions and disincentivizing low-value or frivolous ones. Thus, transaction fees play a vital role in optimizing the use of limited block space and ensuring the economic efficiency of the blockchain network.

11.2.2 Importance of Transaction Fees

Transaction fees play a crucial role in the functioning of blockchain protocols with native cryptocurrencies, such as Bitcoin and Ethereum. These fees serve two primary purposes that are vital for the efficiency and sustainability of the blockchain network.

1. Discouraging Spam and Frivolous Usage
Even if a blockchain protocol had infinite processing power, it would still be essential to impose transaction fees. A very low and nominal fee could be charged per transaction to discourage spam and frivolous usage of the blockchain. Without transaction fees, malicious actors could flood the network with numerous unnecessary transactions, causing congestion and hindering legitimate transactions from being processed promptly.

By charging a small fee, blockchain networks can prevent these types of abusive behaviors, making it economically unfeasible for attackers to conduct large-scale spam attacks. This ensures that the blockchain resources are dedicated to genuine and valuable transactions that contribute to the overall functionality and utility of the network.
2. Prioritizing Valuable Transactions
The limited processing capacity and block size of blockchains pose a challenge in deciding which transactions to

include in each block. With a large number of pending transactions at any given time, miners or validators must choose which ones to include in the next block.

Transaction fees serve as an incentive mechanism for miners to prioritize certain transactions over others. Transactions that offer higher fees are given preferential treatment, as miners are economically motivated to include them in their blocks. This ensures that transactions with a higher perceived value to their creators are processed promptly.

By setting the transaction fees higher for transactions that generate more significant value or have specific urgency, blockchain networks can optimize their throughput and allocate resources more efficiently. This prioritization mechanism aligns intending to maximize the overall utility of the blockchain network.

11.2.3 Setting Transaction Fees: User-Suggested and Protocol-Computed

As we mentioned earlier, transaction fees are a critical component in determining the order in which transactions are included in a blockchain. They serve as an essential mechanism to prioritize valuable transactions while also discouraging spam and frivolous usage of the blockchain. There are two main approaches used to set transaction fees in blockchain protocols with native cryptocurrencies: user-suggested transaction fees and protocol-computed transaction fees.

User-Suggested Transaction Fees
In the user-suggested approach, users themselves can propose the transaction fees they are willing to pay for their transactions to be included in the blockchain. This mechanism is commonly implemented using first-price auctions. Each user attaches a fee to their transaction, indicating the amount they are willing to pay to have their transaction processed quickly.

When a new block needs to be created, miners or validators select transactions to include in the block based on the transaction fees offered. Transactions with higher fees are given priority and are more likely to be included in the next block. Users have an incentive to set competitive fees, as they want their transactions to be processed promptly. At the same time, miners or validators are motivated to include transactions with higher fees, as they receive the fees as rewards for processing those transactions.

This user-suggested approach has been employed in Bitcoin since its inception and was also used in Ethereum until August 2021. Users have the freedom to decide how much they are willing to pay for their transactions to be processed quickly, which can lead to varying transaction fees based on the urgency of the users' needs.

Protocol-Computed Transaction Fees
An alternative approach to setting transaction fees is to have the protocol itself compute the fees based on predefined rules and conditions. Ethereum's new transaction fee mechanism, known as EIP1559, follows this protocol-computed approach.

Under EIP1559, the transaction fees are determined by a formula that takes into account the network's demand and available capacity. The protocol dynamically adjusts the fees based on the congestion in the network. If there is high demand for block space and the network is nearing its capacity, the transaction fees will increase to prioritize transactions. Conversely, when the network is less congested, the fees may decrease, allowing users to enjoy lower transaction costs.

The main advantage of protocol-computed transaction fees is that it provides a more predictable and stable fee structure. Users do not need to guess the appropriate fee to set, as the protocol automatically determines the fee based on the current network conditions. Additionally, this approach can lead to a more efficient allocation of block space, ensuring that transactions with higher value or urgency are prioritized during times of high demand.

In conclusion, setting transaction fees in blockchain protocols with native cryptocurrencies is crucial for managing limited throughput and encouraging efficient usage of the blockchain. Both user-suggested and protocol-computed approaches have their merits, with user-suggested fees providing flexibility and user choice, while

protocol-computed fees offer stability and efficiency in determining fees based on network demand. Each approach caters to different use cases and network requirements, contributing to the overall functionality and scalability of blockchain protocols.

11.3 First-Prize Auctions

In this section, we will explore the most natural and reasonable solution for setting transaction fees in a blockchain protocol: the first price auction. Nakamoto's original proposal for Bitcoin included first price auctions as a transaction fee mechanism, which is still used in Bitcoin to this day. Ethereum also used this mechanism until August 2021 when they switched to a new transaction fee mechanism, which we will discuss later.

11.3.1 Introduction

In a blockchain protocol that uses a first price auction to set transaction fees, users are required to submit their bids along with their transactions. When a user creates a transaction and submits it to the network, they not only provide the necessary information like their digital signature but also include a bid indicating how much they are willing to pay for their transaction to be included in the blockchain and executed.

For example, let's consider a scenario where Alice wants to transfer some native coins from her account A to Bob's account B. When Alice initiates the transaction, she will specify the number of coins to be transferred, sign the transaction with her private key, and include a bid denominated in the blockchain's native currency. This bid represents the amount she is willing to pay to have her transaction processed.

If Alice's transaction is not selected by any miner to be included in a block, she doesn't have to pay her bid. However, if a miner decides to include her transaction in a block and successfully adds that block to the blockchain, Alice will be required to pay her bid to the miner.

Now, the natural question arises: To whom should Alice pay her bid? The answer is straightforward. Since it is the block producer (in a proof of work blockchain, it would be the miner of the block) who decides which transactions to include in the block, it is reasonable for Alice's payment to go to the producer of the block.

So, when Alice's transaction is executed and included in a block, the movement of coins specified in her transaction occurs, and additionally, a portion of her coins goes to the miner of that block. This transfer of the transaction fee bid to the block producer is a crucial incentive mechanism that encourages miners to include users' transactions in the blockchain.

It's important to note that the transfer of the transaction fee bid to the miner doesn't happen immediately. The bid becomes a valid transfer only when the block in which Alice's transaction is included becomes part of the longest chain. This concept is integral to the security of the blockchain, ensuring that only valid and confirmed transactions are rewarded.

Once the block is deeply embedded in the longest chain (often referred to as "confirmed" or having "sufficient confirmations"), the transaction fee bid can be considered permanently transferred to the miner. This confirmation process ensures that other participants in the network recognize the payment and consider the transaction successfully executed.

In summary, a first price auction in a blockchain protocol allows users to submit bids along with their transactions. These bids represent the amount users are willing to pay for their transactions to be included and executed in the blockchain. When a miner includes a transaction in a block, the corresponding bid becomes a valid transfer to the miner, incentivizing them to include more transactions and ensuring the security and efficiency of the blockchain.

11.3.2 Transaction Execution and Revenue

When a transaction is executed in a blockchain protocol that uses a first price auction for transaction fees, two important processes occur:

1. **Coin Transfer:** The transaction creator specifies the movement of coins in their transaction. For instance, if a user wants to perform a simple currency transfer, they would include details on how many native coins they want to move from one account to another (e.g., from Account A to Account B). This transfer is a fundamental aspect of the transaction's purpose.

2. **Transaction Fee Payment:** Along with the transaction details, the creator is also required to provide a bid denominated in the blockchain's native currency. This bid represents the amount the user is willing to pay to have their transaction included in the blockchain and executed. The bid is an incentive for miners, who are responsible for selecting and validating transactions, to prioritize this transaction over others with lower bids.

3. **Payment to Miners:** The bid, which is part of the transaction fee, is directed to the miner of the block that includes the transaction. The block producer decides which transactions are added to their block, and by selecting transactions with higher bids, they can maximize their revenue from transaction fees.

4. **Conditions for Fee Payment:** It's important to note that the transaction fee is only paid if the transaction is successfully included in a block. If, for any reason, the transaction is not selected to be part of the blockchain (e.g., due to network congestion or low bids), the user is not required to pay the fee.

5. **Validation and Confirmation:** In blockchain protocols that utilize a proof-of-work consensus mechanism, such as Bitcoin or Ethereum until August 2021, the validation of transactions and the addition of new blocks occur through mining. Once a block is mined and added to the blockchain, the bids associated with the included transactions become valid and confirmed. However, the bid is considered permanently transferred to the miner only when the block is deeply embedded in the longest chain, usually confirmed by several additional blocks.

6. **Impact on Revenue:** The transaction fee revenue is additive with the block reward revenue for the block producer. The block reward consists of newly minted coins, which are predetermined by the protocol, whereas the transaction fees are a transfer of existing coins from the transaction creators to the miners. Both sources of revenue contributed to the incentive for miners to participate in block production and secure the blockchain.

To illustrate the concept, consider a scenario in which a user wants to transfer 10 native coins from Account A to Account B in a blockchain network that uses a first price auction for transaction fees. The user includes a bid of 5 native coins with the transaction.
If a miner selects this transaction and successfully includes it in a block, two things will happen. First, the 10 native coins will be transferred from Account A to Account B, as specified in the transaction. Second, the bid of 5 native coins will be paid to the miner of the block.
It's important to remember that the bid is not immediately transferred to the miner. The bid's transfer is only confirmed when the block becomes part of the longest chain, validated by additional blocks. Once the block is deeply entrenched in the blockchain, the bid is considered permanently transferred to the miner.
Overall, the first price auction for transaction fees provides a way for users to prioritize their transactions by offering higher bids and enables miners to maximize their revenue by selecting transactions with higher bids for inclusion in blocks.

11.3.3 Distinct Sources of Rewards

In blockchain protocols, there are two distinct sources of rewards for block producers: block rewards and transaction fees. It's crucial to recognize the differences between these two sources, as they have distinct properties and depend on different factors.

Block Rewards

The block reward is a type of reward received by the block producer for successfully mining and adding a new block to the blockchain. It consists of newly minted coins and is a fundamental part of the blockchain protocol. The block reward is predictable and determined by the protocol code. For instance, in Bitcoin, the current block reward is 6.25 Bitcoins per block, and in Ethereum, it is 2 Ether per block.
The block reward is denominated in the native currency of the blockchain, which means that it is specified in the form of the cryptocurrency being mined (e.g., Bitcoins in the case of Bitcoin). However, its USD value can vary over time due to fluctuations in the coin's price in the market. If the price of the native currency rises, the USD value of the block reward also increases. For instance, if the price of Bitcoin doubles, the block reward for successfully mining a block in Bitcoin will also double in USD terms, making it more lucrative for the block producer.

Transaction Fees

Transaction fees, on the other hand, are a separate stream of revenue for block producers and represent the fees paid by users for having their transactions included in a block. Unlike the block reward, transaction fees are not determined by the protocol itself. Instead, they are dictated by the market demand for space on the blockchain.

When a user creates a transaction, they submit a bid denominated in the protocol's native currency, expressing how much they are willing to pay for their transaction to be executed. The bid is essentially a transaction fee that the user offers to the block producer to incentivize the inclusion of their transaction in a block.
However, transaction fees are not directly transferred to the miner when the block is included in the blockchain. Instead, the transaction fee becomes a valid transfer only when the block is deeply entrenched in the longest chain. This ensures that the bid is paid only when the transaction is considered confirmed and final.
The value of transaction fees, in USD terms, is determined by the market for blockchain execution. If there is a high demand for executing transactions on the blockchain, users may be willing to pay higher transaction fees to have their transactions processed quickly. For example, if there is a sudden surge in demand for executing Non-Fungible Token (NFT) transactions on the Ethereum blockchain due to an NFT drop event, users may be willing to pay higher fees to ensure their transactions are included in a block promptly.
It's important to note that transaction fees are conceptually denominated in USD terms, even though they are implemented as native currency transfers. This means that changes in the coin's price will affect the number of native coins required to pay the same USD-denominated transaction fee. For instance, if the price of Ether (ETH) doubles, users will need to pay only half the amount of ETH to achieve the same USD-denominated transaction fee.

In summary, block rewards and transaction fees are two distinct sources of rewards for block producers in blockchain protocols. The block reward consists of newly minted coins and is determined by the protocol code, while transaction fees are dictated by the market demand for blockchain execution. The block reward is denominated in the native currency of the blockchain, whereas transaction fees are conceptually denominated in USD terms. Understanding the differences between these two sources of rewards is crucial for analyzing their impact on blockchain security and stability.

11.3.4 Impact of Transaction Fees on Blockchain Security

In the context of blockchain security, the level and variability of transaction fees play a crucial role. Specifically, high and variable transaction fees can have implications for the security of the blockchain, particularly in relation to the concept of selfish mining attacks.

Recall that transaction fees are determined by the market for space on the blockchain. Users are willing to pay fees in USD terms to ensure that their transactions are executed and included in a block. The market demand for execution on the blockchain dictates the size of transaction fees, which are conceptually denominated in USD terms.

Now, let's consider how these transaction fees can impact the security of the blockchain. Large transaction fees can exacerbate selfish mining attacks, which were discussed in the previous chapter (chapter 10).
In a selfish mining attack, a malicious miner or group of miners seeks to gain an unfair advantage by withholding newly mined blocks from the network. This allows them to secretly mine additional blocks in private, eventually revealing them to the network. As a result, they have a higher probability of extending their chain and potentially reorganizing the blockchain.
The motivation for selfish mining comes from the desire to maximize revenue and obtain a larger share of the block rewards and transaction fees. If transaction fees are high and variable, the incentives for such attacks become stronger.

Let's consider an example to illustrate this point. Imagine a scenario where there is intense competition among users to have their transactions included in the blockchain due to high demand for blockchain resources. As a result, users are willing to pay significant transaction fees in USD terms to have their transactions prioritized by miners.
In this situation, selfish miners can strategically manipulate their mining behavior to prioritize certain transactions over others. By doing so, they can increase their chances of receiving higher transaction fees, thereby maximizing their revenue. This could involve withholding blocks that include lower-paying transactions and only

revealing them when it is most advantageous for the selfish miner's chain to dominate the network.
As transaction fees fluctuate based on market demand, the incentives for selfish mining attacks may vary over time. During periods of high transaction fees, the temptation for malicious miners to engage in selfish mining becomes more pronounced, as the potential rewards are greater.
To address these security concerns, blockchain protocols need to carefully consider the design of their transaction fee mechanisms. Ethereum, for instance, introduced the EIP-1559 transaction fee mechanism to mitigate some of the challenges posed by variable transaction fees and their impact on blockchain security.

Transaction fees, when both generally high and variable, can create stronger incentives for selfish mining attacks, where malicious miners aim to maximize their revenue by selectively including transactions in blocks. This highlights the importance of designing transaction fee mechanisms that strike a balance between user incentives and blockchain security.

In this section, we explored the first price auction as a natural solution for setting transaction fees in blockchain protocols. We discussed the distinct sources of revenue for block producers, the block rewards, and the transaction fees. The difference between the protocol-determined block rewards and market-driven transaction fees is crucial to understand their impact on the blockchain's security and stability. In the section video, we will delve into how large transaction fees can exacerbate selfish mining attacks and then move on to discuss Ethereum's new transaction fee mechanism, EIP-1559.

11.4 Selfish Mining with Transaction Fees

In the last section, we discussed the rewards received by black producers in blockchain networks. Black producers receive rewards from two main sources: block rewards and transaction fees.

As we saw earlier, Block rewards are newly minted coins given to the miner who successfully mines a block. Historically, block rewards have been significantly larger than transaction fees in most blockchain networks. For example, in the Bitcoin blockchain, block rewards have always been much larger than transaction fees.

Transaction fees are the fees paid by users for their transactions to be included in a block. They are transferred to the miner who successfully mines the block. In the past, transaction fees were dwarfed by the value of block rewards, but this has changed in recent times.

11.4.1 Future Projections

Looking ahead, there are possible future scenarios regarding the dominance of block rewards and transaction fees in blockchain networks. In both Bitcoin and Ethereum, the dynamics of these rewards are subject to change due to various factors.

Ethereum Case

For Ethereum, initially, transaction fees were much smaller than block rewards. However, with the rise of decentralized finance (DeFi) applications, the demand for the Ethereum blockchain increased significantly. As a result, transaction fees on the Ethereum network have increased significantly in recent times. In some cases, the transaction fees have even exceeded the block rewards.
This surge in transaction fees is attributed to the growing popularity of DeFi platforms and the high demand for computational resources on the network. If this trend continues, there is a possibility that transaction fees might eventually dominate block rewards in Ethereum. Depending on the continued growth of demand and the price of ether, transaction fees could potentially dwarf the block rewards in the future.

Bitcoin Case

In the case of Bitcoin, historically, block rewards have consistently been much larger than transaction fees. However, it emphasizes that the situation could change in the future. This is because the block reward in Bitcoin undergoes periodic halving events approximately every four years. The block reward, which started at 50 bitcoins, has been halved multiple times, and currently stands at 6.25 bitcoins. This halving process will continue until

approximately the year 2140 when the block rewards will effectively become zero. As a result, the dominance of block rewards in Bitcoin is expected to diminish over time. If the demand for the Bitcoin protocol remains steady, transaction fees could potentially surpass block rewards in the distant future.

Implications

If transaction fees start to surpass block rewards in either Ethereum or Bitcoin, it could lead to significant implications for the blockchain networks. One key concern is the potential increase in selfish mining attacks. These attacks involve miners deliberately creating forks in the blockchain to maximize their rewards.

In the current scenario, where transaction fees are not substantially higher than block rewards, there is less incentive for miners to engage in selfish mining attacks. However, if transaction fees become significantly larger than block rewards, even smaller miners may find it financially appealing to attempt these attacks.
The consequence of increased selfish mining attacks is a potential slowdown in the progress of the blockchain network. If a significant portion of miners is focused on forking the chain to steal each other's rewards, the overall efficiency of the blockchain can be compromised, and the network's security may be at risk.
While selfish mining attacks have not been a major issue in Bitcoin so far, the impending halving events that decrease block rewards may change this dynamic. Similarly, in Ethereum, the rising popularity of DeFi applications has driven up transaction fees, making selfish mining attacks more tempting for certain miners.
To address these potential concerns, blockchain networks should actively monitor the relationship between block rewards and transaction fees and consider implementing mechanisms to deter selfish mining attacks. Ethereum's EIP-1559, for instance, introduces a new transaction fee mechanism that unintentionally helps mitigate the severity of such attacks by reducing overall miner rewards.

EIP-1559

We wish to briefly explain EIP-1559, a proposal for Ethereum that introduces a new transaction fee mechanism. Although the primary motivation for EIP-1559 is not directly related to mitigating selfish mining attacks, the new mechanism inadvertently addresses this issue to some extent. By burning a portion of the transaction fees, EIP-1559 reduces the overall rewards received by miners, making selfish mining attacks less attractive.

11.4.2 Selfish Mining Attacks

Selfish mining attacks are deliberate attempts by miners to create forks in the blockchain, aiming to maximize their own rewards. In the previous chapter, we discussed how the profitability of these attacks depends on several factors.

Factors Influencing Profitability

The profitability of selfish mining attacks hinges on two key factors:

1. **Tie-Breaking Assumptions:** When there is a tie in the longest chain, it is crucial to consider what assumptions are made about how honest nodes choose between competing chains. If an attacker can influence tie-breaking decisions, selfish mining becomes more profitable.

2. **Hash Rate of the Attacker (α):** The hash rate of the selfish miner relative to the total network hash rate plays a significant role in determining the success of the attack. The higher the attacker's hash rate, the more advantageous selfish mining becomes.

In the best-case scenario, where honest nodes always favor the chain with an honest block when there is a tie, selfish mining attacks are less effective. However, even with best-case tie-breaking, if the attacker possesses more than one-third of the total hash rate ($\alpha > \frac{1}{3}$), selfish mining can still be profitable.

11.4.3 Large Transaction Fees

Here, we discuss a scenario where transaction fees in a blockchain network are significantly higher than the average reward received from mining a block, including both the block rewards and transaction fees. The focus

is on understanding how this situation may create an incentive for miners, even those with relatively small hash rates, to engage in selfish mining attacks.

> **Example 11.4.1** (Suppose we are in a blockchain network where transaction fees are, on average, ten times higher than the block reward. Let's consider a typical block that has 10 times the value of transaction fees compared to the block reward. In this hypothetical case, the block rewards remain relatively steady as they are protocol-computed, while transaction fees vary based on demand.)

Now, imagine a situation where the most recent block added to the end of the longest chain (let's call it B_3) contains not just typically high transaction fees but exceptionally high ones, let's say they are 100x times the block reward value. This makes B_3 a block with super high transaction fees. (Figure 11.1)

Miner's Decision: Now, if a miner has a relatively small hash rate (for example, 1% of the total network hash rate), they face a tempting decision. As an honest miner, they should ideally mine on the end of block B_3 and add new blocks to extend the chain with high transaction fees. However, given the extremely high transaction fees in B_3, the miner might be tempted to attempt a selfish mining attack.

Instead of extending B_3, the miner may choose to mine a new block (let's call it B'_3) that has B_2 as its predecessor. By doing so, the miner is effectively forking the blockchain and attempting to steal the super high transaction fees for themselves. In this case, B'_3 is a block created by the miner with their public key as the miner address.

Probability of Success: Suppose the miner has a hash rate of 1%, meaning they have a one in five chance of being the next one to produce a block. If the miner successfully mines B'_3 and extends it by mining a new block B'_4, they will have successfully orphaned block B_3 and taken the super high transaction fees for themselves.

However, there's a risk involved, as there's a four out of five chance that another miner produces a block extending B_3 before the miner can successfully mine B'_4. In this case, the miner who attempted the selfish mining attack would lose the block reward and transaction fees they could have earned by honestly mining on the main chain.

Expected Benefit Despite the risk, the miner may still decide to proceed with the selfish mining attack. With a 1% hash rate, the expected benefit of successfully orphaning block B_3 and claiming the high transaction fees outweighs the expected loss from four out of five times when the attack fails.

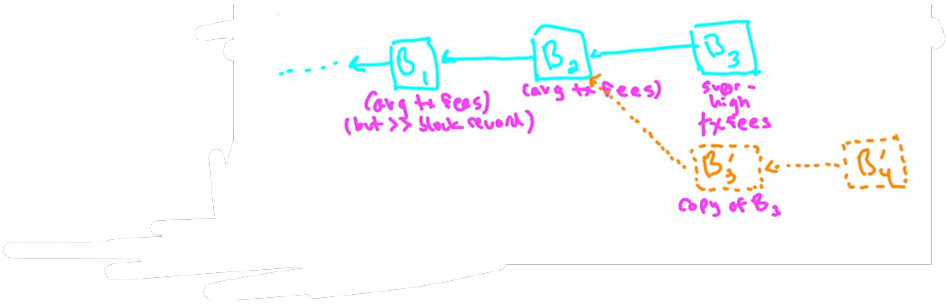


Figure 11.1: Large Transaction Fees: example

Implications

Even with a small hash rate (1% in the example), there can be situations where miners find it profitable to engage in selfish mining attacks if the transaction fees in a particular block are much higher than the average reward from a block.

Addressing the Issue

This situation raises concerns about the incentives provided to miners in the presence of large transaction fees. In the extreme case where multiple miners attempt selfish mining attacks to steal each other's rewards, progress in the blockchain could slow down significantly or even lead to a lack of liveness. Such issues are not currently prevalent in Bitcoin, it is essential to consider the future as block rewards are halved periodically. Over time,

the dominance of transaction fees over block rewards may increase, potentially leading to increased incentives for selfish mining attacks.
As we mentioned, Ethereum's transition to EIP-1559 introduces a new transaction fee mechanism that mitigates the severity of selfish mining attacks. The mechanism involves burning some transaction fees, reducing the amount received by miners and, consequently, reducing the incentive for selfish mining attacks.

11.5 Issues with First-Price Auctions

In this section, we will discuss the new transaction fee mechanism proposed in EIP-1559 for Ethereum. Before delving into the details of how it works, we will first explore the need for an alternative to first-price auctions (FBAs) and the issues associated with them. Despite the success of FBAs in Bitcoin and Ethereum, there are significant challenges in figuring out how to bid optimally in such auctions.

11.5.1 Challenges of First-Price Auctions

First-price auctions are widely used in various application domains, including blockchain systems like Bitcoin and Ethereum. However, despite their prevalence, they come with significant challenges, particularly in the context of determining optimal bidding strategies. Let's delve into the main issues associated with first-price auctions and explore why they may not be the most suitable option for transaction fee mechanisms.

Uncertainty of Competing Bids

One of the primary difficulties in first-price auctions is the uncertainty surrounding the bids of other participants. Imagine you are a user who wants to include a transaction in a block, and the transaction fee is determined through a first-price auction. If you had telepathy and could magically know all the other participants' bids, your decision-making process would be straightforward. You could bid just high enough to outcompete others and secure a spot in the block. Unfortunately, in reality, this is not possible, and participants must make educated guesses about what their competitors might bid.

Educated Guesses and Regretting Bids

Making these educated guesses is no simple task, even for the most sophisticated individuals. Regardless of your intelligence and the quality of your guesses, there's always a chance that you will regret your bid in hindsight. For instance, consider the following scenarios:

Scenario 1: Bidding Too High
You decide to bid a relatively high amount, hoping to secure your transaction's inclusion in the block. Eventually, your bid wins, and your transaction is added. However, upon reflection, you realize that you overpaid for the transaction to be included. In this case, you regret not having bid a lower amount.

Scenario 2: Bidding Too Low
Alternatively, you may decide to bid conservatively, hoping to save on transaction fees. However, in doing so, you lose the bidding competition, and your transaction is excluded from the block. You discover that the bids were not as high as you initially assumed, and your conservative bidding strategy backfired. Consequently, you regret not having bid more aggressively.

Automated Bidding and Its Limitations

Some users opt to rely on software wallets that automate the bidding process based on recent bid data from the blockchain. While this is a step towards simplifying the process, it does not guarantee optimal bids. Users may still face frustration if their automated bids underestimate the optimal bid, leading to prolonged transaction confirmation times.

User Experience on Public Blockchains

Regular users of public blockchains like Ethereum often encounter issues with transaction fees and bid estimations. Users submit transactions through their wallets, which suggest appropriate bid amounts. However, due to the

challenges mentioned earlier, there are instances where wallets underestimate the required bid, leading to delays in transaction confirmations. As a result, users may experience transaction hang times of significant duration, causing frustration and dissatisfaction.

The challenges of first-price auctions lie in the difficulty of optimally determining bid amounts without knowing the competing bids. This uncertainty can lead to regrets about both overpaying and underbidding, making the process less user-friendly and reliable. As a response to these challenges, Ethereum introduced the EIP-1559 mechanism, which aims to address these issues and create a more efficient and user-friendly transaction fee mechanism.

11.5.2 Introducing EIP-1559

The solution to the challenges of first-price auctions was proposed in Ethereum Improvement Proposal (EIP) 1559. This new mechanism introduced a fixed "base fee" that is automatically adjusted based on network demand. Users would now only need to include the base fee with their transactions, and the network would handle the rest.

With EIP-1559, the process of bidding on transaction fees became much simpler and more akin to the take-it-or-leave-it model of online shopping. Users no longer needed to make complex calculations or educated guesses about competing bids. Instead, they only needed to accept the prevailing base fee, which would be dynamically adjusted to reflect network conditions.

By adopting EIP-1559, Ethereum aimed to make the bidding process for transaction fees more user-friendly and efficient, reducing the frustration users faced with the old first-price auction mechanism. This change represented a significant step towards improving the overall user experience on the Ethereum network.

11.5.3 Second Price Auctions (Vickery Auctions)

In the context of alternatives to first-price auctions, we can name Second Price Auctions, also known as Vickery Auctions. These auctions offer a dominant strategy for bidders, meaning it is always in their best interest to bid their maximum willingness to pay.

Dominant Strategy in Vickery Auctions

In a Vickery Auction, the auction mechanism itself optimally shades the bids on behalf of the bidders, given the bids of all competitors. This means that a bidder should place the highest price they are willing to pay for an item, and the auction will automatically ensure they pay just one increment above the second-highest bid. As a result, bidders have no incentive to shade their bids, unlike in first-price auctions where bidders tend to bid less than their maximum willingness to pay to avoid overpaying.

Comparison to First-Price Auctions

While Second Price Auctions offer a straightforward bidding strategy, they are not suitable for blockchain applications like Bitcoin and Ethereum, specifically when it comes to transaction fees. The main issue is that these auctions rely on trusting the auctioneer or whoever runs the auction not to manipulate the price by injecting additional bids. In the case of blockchain, where miners have significant control over the blocks they produce, such manipulatability can lead to abuse and unfair practices.

Inappropriateness for Blockchain Deployment

Using Second Price or Vickery Auctions in the context of blockchain transaction fees would incentivize miners to create new transactions and include them in the block to boost their revenue from these auctions. This undermines the integrity of the blockchain system and is one of the reasons why such auction mechanisms are not used for determining transaction fees.
While Second Price Auctions offer an appealing dominant strategy for bidding, they are not a suitable solution for determining transaction fees in blockchains like Ethereum due to the potential for manipulation and abuse by miners. Instead, the EIP-1559 mechanism was introduced to provide an alternative that simplifies the bidding process while addressing the specific challenges faced in blockchain environments.

11.6 EIP-1559

In this section, we will explore the transaction fee mechanism proposed in EIP-1559, which was developed by Vitalik Buterin, the founder of Ethereum. The EIP stands for Ethereum Improvement Proposal. The new mechanism was introduced to address the drawbacks of first-price auctions, which were previously used for setting transaction fees in blockchain protocols like Bitcoin and early versions of Ethereum.

Intro. First-price auctions were a natural choice for transaction fees in blockchain protocols. However, they had a significant drawback – it was challenging for users to bid intelligently in such auctions. This issue was highlighted by the fact that users needed to consider their competitors' bids and make educated guesses about them. Moreover, there was a constant risk of either overpaying for transaction inclusion or under-bidding and missing out on being included in a block. This contrasted sharply with a more familiar scenario like shopping on Amazon, where users could simply decide whether a product's price was acceptable to them and make a straightforward choice.

The Need for a Better Transaction Fee Mechanism:
To illustrate this drawback further, imagine a first-price auction scenario in the blockchain context. Users would submit their bids for transaction fees, and the miner would select the transactions with the highest bids for inclusion in the block. However, since users do not have perfect information about their competitors' bids, they might end up bidding either too high or too low, leading to suboptimal outcomes.

For example, let's say there are three pending transactions, and users A, B, and C submit bids of 20, 25, and 30 units of the native cryptocurrency, respectively. The miner, who is responsible for selecting the transactions for the block, chooses transaction C with the highest bid of 30 units. However, in this scenario, if user A had known that user B's bid was 25 units, they could have bid 26 units to get their transaction included instead. Similarly, if user C had known about user A's bid, they might have bid just enough to outbid user A, say 21 units. In this way, the first-price auction mechanism leads to a lack of transparency and suboptimal outcomes for users.

This issue becomes even more challenging when considering the dynamic nature of transaction fees in a blockchain. As the number of transactions and the network's demand fluctuates, so do the fees. Users have to constantly adapt to changing conditions and make guesses about other users' behavior to bid effectively.

The desire for a more user-friendly and predictable transaction fee mechanism, akin to shopping on Amazon, led to the development of the transaction fee mechanism proposed in EIP-1559.
EIP-1559 introduces the concept of a base fee, which serves as a reserve price or minimum bid for transaction inclusion in a block. Transactions with bids lower than the base fee are considered invalid and cannot be included in the block. This base fee is denominated in units of the native cryptocurrency, making it easy for users to understand. Additionally, users have the option to bid higher than the base fee if they wish to increase their chances of inclusion in a block.

The key idea behind EIP-1559 is to make the fee mechanism more transparent and user-friendly. Users no longer have to worry about outbidding their competitors or making complex decisions based on incomplete information. Instead, they can focus on whether the base fee is acceptable to them, just like a straightforward take-it-or-leave-it offer. This simplicity and predictability bring the user experience closer to that of shopping on Amazon, where users can decide whether a product's price is worth it to them without the need for complex calculations or competition analysis.

Note:-

Key Idea #1

Each block has a base free $r > 0$(reserve price)

- minimum bid to be eligible for inclusion (it's okay to bid more)
- Deterministic (protocol-computed) function of predecessor blocks (can vary with the block)

The mechanism's success lies in the combination of the base fee and the fee-burning aspect. By burning

the base fee revenues instead of passing them on to the miner, collusion between users and miners becomes ineffective. This ensures that users cannot manipulate the system to their advantage and promotes fair and efficient transaction inclusion.

11.6.1 Determining the Base Fee

The base fee is a critical component of the transaction fee mechanism proposed in EIP-1559. It acts as a reserve price or minimum bid for transaction inclusion in a block. Any transaction with a bid lower than the base fee is considered invalid and cannot be included in the block. This ensures that all transactions included in a block meet a minimum fee requirement.

The base fee is denominated in units of the native cryptocurrency, which makes it easy for users to understand the minimum fee they need to pay for their transactions to be eligible for inclusion.

Deterministic Computation

One essential characteristic of the base fee is that it is protocol-computed. This means that it is not suggested by users like in first-price auctions. Instead, the base fee is determined entirely by the protocol, based on the history of past blocks. It is a deterministic function of everything that has happened leading up to the current block. The contents of the current block do not influence the computation of the base fee.
This deterministic computation ensures that the base fee remains objective and transparent. Users do not have to worry about subjective biases or manipulations in setting the base fee. It is purely based on historical data and is uniquely determined by past events.

Variability with Blocks

The base fee is not fixed but can vary from block to block. As the blockchain's demand and transaction volume fluctuate, the base fee can adjust accordingly. For example, if there is a surge in network activity, the base fee might increase to incentivize miners to include more transactions in the block. Conversely, during periods of lower activity, the base fee may decrease to reflect the reduced demand.
This variability helps the mechanism adapt to changing network conditions and ensures that the transaction fee system remains efficient and responsive to demand fluctuations.

Example of Base Fee Adjustment

To illustrate the concept of base fee variability, let's consider a hypothetical example. Suppose the blockchain has been operating with a base fee of 100 units of the native cryptocurrency for several blocks. However, due to increased network activity and demand, the protocol computes the base fee for the next block to be 120 units.
In this scenario, any transaction with a bid lower than 120 units will be considered invalid and ineligible for inclusion in the next block. Users who want their transactions to be included must bid at least 120 units, which becomes the minimum bid for this specific block.
Conversely, in the subsequent block, if the network activity decreases, the protocol might adjust the base fee to 90 units. Now, transactions with bids lower than 90 units will be invalid for inclusion in that block.
This dynamic adjustment of the base fee based on network conditions ensures that the transaction fee mechanism remains responsive and efficient, promoting optimal transaction inclusion in blocks.

11.6.2 Destination of Base Fee Revenues

In the context of EIP-1559, the destination of base fee revenues plays a crucial role in determining the effectiveness of the transaction fee mechanism. Initially, it may seem reasonable to direct the base fee revenues to the miner of the block, just like in first-price auctions. After all, the miner is the one who decides which transactions to include in their block, so it makes sense to reward them with transaction fees.

However, the text highlights a key problem with this approach. If the base fee revenues are passed on to the miner, it essentially renders the base fee useless. Users and miners could potentially collude to avoid paying the base fee and manipulate the system to their advantage.

To illustrate this issue, let's consider the following example:

In this collusion scenario, users would publicly bid r (the base fee) for their transactions on-chain, making it seem like they are willing to pay the full base fee. However, they would privately communicate with the miner through a side channel, indicating that they are only willing to pay b_i, for instance here, 45 (less than the base fee). The miner, being part of this collusion, would include all these transactions in the block, collecting the base fee of 100 from each of them. Off-chain, the miner would then refund the difference $r - b_i$ for example here (100 - 45 = 55) back to the users, essentially nullifying the base fee.

As a result, none of the pending transactions would be eligible for inclusion in the block, and the miner would be forced to produce an empty block. While the miner would still collect the block reward, they would not receive any transaction fee revenue, leading to an undesirable outcome for both users and miners.

To circumvent this situation and achieve better game theoretic properties, EIP-1559 proposes an alternative approach for the destination of base fee revenues: fee burning. Instead of rewarding the miner with the base fee revenues, the mechanism simply burns the fees by sending them to a null address, effectively taking those coins out of circulation forever.

Example 11.6.1 (The base fee (r) is set at 100 units of the native currency. Now, imagine there are several pending transactions, and all of them are only willing to pay up to a maximum of 50 units as their bid. In this scenario, if all participants act honestly, they would bid at most $r/2$, i.e., 50 units, since that is what they are willing to pay.
In a scenario where the base fee is not burned, users and the miner could potentially collude to manipulate the system and pay less than the base fee while still getting their transactions included.
In this collusion scenario, users would publicly bid r (the base fee) for their transactions on-chain, making it seem like they are willing to pay the full base fee. However, they would privately communicate with the miner through a side channel, indicating that they are only willing to pay b_i, for instance here, 45 (less than the base fee). The miner, being part of this collusion, would include all these transactions in the block, collecting the base fee of 100 from each of them. Off-chain, the miner would then refund the difference $r - b_i$ for example here (100 - 45 = 55) back to the users, essentially nullifying the base fee.

As a result, none of the pending transactions would be eligible for inclusion in the block, and the miner would be forced to produce an empty block. While the miner would still collect the block reward, they would not receive any transaction fee revenue, leading to an undesirable outcome for both users and miners.

To circumvent this situation and achieve better game theoretic properties, EIP-1559 proposes an alternative approach for the destination of base fee revenues: fee burning. Instead of rewarding the miner with the base fee revenues, the mechanism simply burns the fees by sending them to a null address, effectively taking those coins out of circulation forever.)

This fee-burning process helps offset the inflation caused by block rewards. In the example given earlier, when some transaction fees are burned, the effective inflation rate of the native currency decreases from 4% to 2%. This means that the value of the native currency held by users is being devalued at a slower rate, providing a financial benefit to all holders.
As a result, the fee-burning aspect of EIP-1559 became a significant selling point for the proposed transaction fee mechanism. It not only addressed the issue of collusion but also presented an opportunity for users to benefit from a reduction in the effective inflation rate, making it appealing to the Ethereum community and gaining widespread support. By opting for fee burning instead of rewarding the miner, the mechanism ensures better game theoretic properties and provides financial benefits to all holders of the native currency.

11.6.3 Excess Transaction Fees

In the new transaction fee mechanism introduced by EIP-1559, users have the option to bid more than the base fee if they wish to increase the chances of their transactions being included in a block. The base fee acts as the minimum bid necessary for eligibility, but any higher bid is also considered for inclusion.

Let's revisit the previous example of collusion between users and miners, where users bid the base fee on-chain but privately communicate to the miner their willingness to pay a lower amount. In this case, the miner includes all transactions with the on-chain bid of the base fee, making them eligible for inclusion. However, since the base fee is not transferred to the miner, they cannot provide refunds off-chain. Therefore, this type of collusion becomes ineffective, and all transactions remain eligible, defeating the purpose of the base fee.
On the other hand, in the presence of honest transactions that bid more than the base fee, the excess transaction fees above the base fee are indeed transferred to the miner of the block. This ensures that miners are incentivized to include transactions with higher fees, as they will receive additional revenue for doing so.

The ability for users to bid more than the base fee allows for greater flexibility in transaction pricing. For example, if a user urgently needs their transaction to be included quickly, they can bid a higher fee to prioritize their transaction over others. Conversely, users with less urgency can choose to bid closer to the base fee to save on transaction costs.

Incorporating excess transaction fees into the mechanism ensures that the base fee does not discourage users from providing higher fees when they find it necessary or beneficial. This way, the mechanism strikes a balance between predictable pricing through the base fee and market-driven pricing with the inclusion of excess fees.

It's worth noting that this combination of base fee, fee burning, and excess transaction fees provides a more efficient and user-friendly transaction fee mechanism compared to traditional first-price auctions. The mechanism incentivizes miners to include transactions with higher fees while reducing the overall inflation rate through fee burning. This aspect of EIP-1559 garnered significant support from the Ethereum community, making it a prominent reason for the adoption of the new transaction fee mechanism.

11.6.4 OCA Proofness Property

Theorem 11.6.1

Miner-user collusion no longer helps.

The Off-Chain Agreement (OCA) proofness property is a critical aspect of the EIP-1559 transaction fee mechanism. It ensures that collusion between users and miners does not lead to any advantages for them, and they cannot manipulate the mechanism off-chain to gain better outcomes than honest participation on-chain.

To better understand the significance of OCA proofness, let's revisit the example of collusion that was mentioned earlier. Suppose we have a block with a base fee of r, set at 100. There are several pending transactions, and all of them are willing to pay up to a maximum of 50. In a scenario where the base fee is not burned, users and the miner could collude to game the system and pay less than the base fee while still getting their transactions included.

In the collusion scenario, users would publicly bid r (the base fee) for their transactions on-chain, but they would inform the miner privately through a side channel that they are only willing to pay 45 (less than the base fee). The miner, in turn, would include all these transactions in the block, collecting the base fee of 100 from each of them. Off-chain, the miner would then refund the difference (100 - 45 = 55) back to the users, essentially nullifying the base fee.

However, with the OCA proofness property in EIP-1559, this collusion strategy proves to be ineffective. The key reason behind this is that the base fee revenues are burned instead of being passed on to the miner. As a result, the miner is not in a position to refund the users since any refund would be an out-of-pocket expense.

In a more general context, the OCA proofness property ensures that no matter what strategies users and miners employ off-chain, they cannot achieve better outcomes than if they participate honestly on-chain. Any attempts to manipulate the mechanism by shading bids, sharing information, or collusion are futile, as the protocol's rules are designed to prevent such efforts.

By satisfying the OCA proofness property, EIP-1559 ensures fairness, transparency, and predictability in the transaction fee mechanism. Users can confidently bid based on their willingness to pay, knowing that the mechanism cannot be exploited or influenced by off-chain agreements.

11.7 What should the Base fee Be?

11.7.1 Ideal Base Fee: Market Clearing Price

The ideal base fee, from an economic efficiency standpoint, would be a *market clearing price*. In this context, a market clearing price refers to a base fee that is set at a level where the subset of pending transactions willing to pay this fee would exactly fill up a block, neither more nor less.

To understand why a market clearing price is considered the best base fee, let's examine the advantages it offers:

1. **100% Utilization:** Setting the base fee at the market clearing price ensures that the blockchain's capacity is fully utilized. All available block space is occupied by transactions, maximizing the usage of this scarce resource, which is the blockchain.
2. **Highest Value Transactions:** The transactions included in the block under a market clearing price are precisely the ones willing to pay this fee. As a result, only the highest-value transactions are included, prioritizing those that offer the most significant value to users.

Consider this example:

Example 11.7.1 (If a magical base fee of 50 units fell from the sky, and there are pending transactions willing to pay up to 50 units, then exactly the right number of transactions to fill up the block would be included. Any additional transactions would be excluded because they are not willing to pay the base fee, and any fewer transactions would indicate that some available block space is unused.)

This concept of a market clearing price aligns with the notion of economic efficiency, where the scarce resource (block space in this case) is utilized optimally and allocated to the highest value transactions. It ensures that the blockchain operates at its full potential, and only the most valuable transactions get included.

A base fee set higher than the market clearing price would be inefficient because the block would not be completely full. In this scenario, the number of transactions willing to pay the higher base fee may not be sufficient to fill up the entire block. On the other hand, setting the base fee lower than the market clearing price would lead to inefficiencies as low-value transactions and high-value transactions would be on equal footing, potentially occupying valuable block space. Therefore, the market clearing price is an important benchmark for the ideal base fee, as it ensures both optimal resource utilization and prioritization of valuable transactions, ultimately contributing to economic efficiency on the blockchain.

11.8 Computing the Base Fee: Variable Size Blocks

Computing the Base Fee in Ethereum's EIP-1559 is done using the concept of **Variable Size Blocks**. This approach allows for the automatic adjustment of the base fee based on the previous block's utilization, which helps to maintain a market clearing price for transactions.

Variable Size Blocks Explained: Traditionally, blockchains have had a hard cap on block size, where each block must not exceed a certain size (e.g., 15 million units of gas). This was done to ensure that all nodes in the network can keep up with the processing of transactions and to maintain minimal computational and storage requirements for running the protocol.

However, EIP-1559 introduces a different approach by introducing a *target block size*. Instead of strictly adhering to a fixed block size cap, the protocol allows for some flexibility. The target block size is set as the desired average size of blocks and is used to bind the block size on average.

Note:-

Key Idea #2

Allow variable size blocks (i.e relax per-block hard cap)

- Per-block size constraint overkill, the average-size constraint should be good enough
- Set target block size C
- Allow blocks of size up to $2C$ (borrow capacity from the near future)

Example of Target Block Size: In Ethereum, if the target block size is set to 15 million units of gas, the average size of blocks should ideally be around this value.

Adjusting the Block Size: The EIP-1559 allows for violations of the target block size within certain limits. In Ethereum, blocks are allowed to be double the target block size, i.e., up to 30 million units of gas.

Benefits of Variable Size Blocks: The introduction of variable-size blocks offers several advantages:

- *Flexibility:* The blockchain can adapt to periods of high demand by borrowing space from the future, allowing it to handle sudden spikes in transaction volume.
- *Efficiency:* The blockchain can optimize block space allocation, ensuring that it remains close to its target block size, leading to more efficient resource utilization.

Local Search for Base Fee Adjustment: The variable size blocks enable the implementation of the local search approach to adjust the base fee.

- If the previous block's size is less than the target block size (an *empty block*), it indicates that the base fee is too high, as there weren't enough eligible transactions to fill the block. In this case, the base fee is decreased by 12.5% for the next block.
- On the other hand, if the previous block's size is double the target block size (a *double full block*), it means there is excess demand at the current base fee level, and the base fee should be increased. Therefore, the base fee is increased by 12.5% for the next block.
- For block sizes that are between the empty and double full conditions, the base fee adjustment is done through *linear interpolation*. This ensures a smooth and gradual adjustment of the base fee based on the deviation from the target block size.

Example of Base Fee Adjustment: Let's say the target block size is 15 million units of gas. If a block has a size of 12 million units of gas (less than the target), the base fee for the next block will be decreased. Conversely, if a block has a size of 30 million units of gas (double the target), the base fee for the next block will be increased.

Variable size blocks play a crucial role in the EIP-1559 transaction fee mechanism, enabling automatic adjustment of the base fee based on the actual demand for block space. By borrowing space from the future when necessary, the mechanism achieves efficient resource utilization and maintains a market clearing price, making it a powerful and user-friendly solution for transaction fee computation in Ethereum.

11.9 Excessively Low Base Fees

Up till now, we discussed first-price auctions and their drawbacks. It is challenging to determine how to bid in such auctions, requiring significant cognitive effort. This led to the introduction of the new transaction fee mechanism proposed in EIP-1559, which Ethereum switched to in early August 2021. The mechanism involves a burned base fee and a local search procedure to adjust the base fee over time. To accommodate sudden spikes in demand, variable-sized blocks were introduced, with a target average block size of C and a hard per-block cap of $2C$.

11.9.1 Two Regimes of the Transaction Fee Mechanism

The transaction fee mechanism proposed in EIP-1559 exhibits two distinct regimes that determine how easy it is for users to figure out how to bid in the Ethereum network.

Definition 11.9.1

A block's base fee r is *excessively low* if the total size of all pending eligible (i.e, bid $\geqslant r$) transactions is $> 2C$(size of a double-full block)

Regime 1: Base Fee Not Excessively Low

In this first regime, the base fee is not excessively low, and there is enough room in the blocks to accommodate all eligible transactions without competition. Bidding in this regime is as simple as shopping on Amazon. Users can estimate the appropriate fees by just bidding the base fee (r) or adding a small tip (Δ) to incentivize miners to include their transactions.

Theorem 11.9.1

If base fee r is not excessively low, it's optional to bid r (or maby $r \times s$ for a small s) so that miners incentivized to include txs rather than produce empty blocks. [Otherwise, reverts to a first-prize auction]

We can use this scenario to go shopping on Amazon, where products have a posted price, and users can take it or leave it. Similarly, in this regime, the blockchain offers a take it or leave it deal for transaction inclusion. Users can bid the base fee or add a tip if they wish, but there is no need for aggressive bidding or complex strategies because there is enough capacity to include all transactions.
Additionally, the excess between the bid and the base fee is burned, meaning that if a user bids more than the base fee, the additional amount is transferred to the miner, and the base fee itself is burned.

This regime is the more common one and is characterized by blocks that are not double full, indicating sufficient space for all eligible transactions. In practice, it is observed that most blocks fall under this regime, providing a straightforward bidding experience for users.

Regime 2: Base Fee Excessively Low

In the second regime, the base fee becomes excessively low, meaning that there is not enough space in the blocks to accommodate all eligible transactions. This situation occurs less frequently, but it does happen in specific scenarios.

When the base fee is excessively low, miners must choose which transactions to include in the limited block space, akin to a first-price auction. This creates a situation where users have to compete with each other through their bids to have their transactions included in the block.

In this regime, the excess between the bid and the base fee is once again burned, but the miner receives the burned amount. As a result, the miner has an incentive to prioritize transactions with higher bids to maximize their share of the transaction fee.
While this situation does arise occasionally, it is not as problematic as traditional first-price auctions. The EIP-1559 mechanism still provides improvements, as the base fee is burned, reducing the risk of strategic manipulation, and there are fewer scenarios where users need to compete intensely.

11.9.2 Determining the Regime

The determining factor for the regime is whether the current base fee (r) is excessively low or not. If the total size of eligible transactions at that base fee exceeds $2C$, it is considered excessively low. Moreover, if it exceeds C, it is already too low, indicating a need to increase it.

11.9.3 Factors Affecting the Regime

Now the important question is, Why might the base fee be excessively low??
The regime, which determines whether the transaction fee mechanism behaves as a posted price mechanism (good regime) or a first-price auction (bad regime), can be influenced by the following factors:

Sudden Increases in Demand

One of the factors that can lead to the bad regime is sudden sharp increases in demand for the blockchain. These spikes in demand often occur during events such as major NFT drops on Ethereum. When the demand for transactions surges, the market clearing price can rise significantly. However, due to the design of the transaction fee mechanism in EIP-1559, the adjustment of the base fee is limited by the protocol.

The local search procedure used in EIP-1559 aims to find the market clearing price with respect to a normal block size (C). However, if there is a sudden increase in demand, the market clearing price can quickly jump to a higher level, exceeding the current base fee (r). As a result, the base fee might be too low to accommodate all the eligible transactions, causing the regime to shift to the bad regime.

During this transitory period, which may last for several minutes, the base fee has not yet caught up with the higher market clearing price. As a consequence, miners may find themselves with an excessively low base fee for a short period, leading to competition among transactions and a first-price auction-like behavior.

Random Variability in Block Production

Another factor that can impact the regime is the random variability in block production, or in other words, unusually long period of time between consecutive blocks, which is characteristic of Ethereum's proof-of-work implementation. While the average block production time is around 13 seconds, it is not precise and can vary. Miners must solve a cryptographic puzzle to produce a block and sometimes it can take longer than usual to find a valid solution.

When blocks take longer to produce than expected, more eligible transactions can accumulate during that time. This can lead to a situation where there are more transactions than the base fee (r) can accommodate, pushing the regime to the bad regime. For example, if the base fee was set to handle the average block production time (13 seconds), but a block takes double the expected time (26 seconds), there will be three times as many eligible transactions as anticipated. This sudden increase in the number of eligible transactions can exceed the capacity of the current base fee, resulting in the regime shifting to the bad regime.

11.9.4 Future Predictions

Currently, based on data from watchtheburn.com, the good regime (base fee not excessively low) is more prevalent than the bad regime (base fee excessively low). It is estimated that the good regime occurs around 90% of the time. However, it is worth noting that Ethereum is planning to switch to a proof-of-stake design in 2022. With proof of stake, the random variability in block production will be reduced since blocks will be produced more regularly, approximately every 12 seconds. As a result, the second culprit, random block time variability, will have less influence on the regime.

After the switch to proof of stake, it is expected that the good regime will become even more prevalent, occurring around 95% of the time, with the bad regime becoming less frequent. Nevertheless, the occasional sudden spikes in demand will still lead to transitory periods where the base fee might temporarily be excessively low.

11.10 Pros and Cons of Burning Fees

11.10.1 Consequences of Burning Base Fee Revenues

The burning of base fee revenues in the EIP-1559 proposal has multiple implications, impacting both miners and holders of the native currency, such as Ethereum. Let's delve into the consequences of this design choice:

Benefiting Holders of Native Currency

By burning the coins used to pay the base fee, these coins are permanently removed from circulation. This reduction in the coin supply has significant implications for the holders of the native currency. In analogy to the equities world, this process resembles stock buybacks, which, in principle, should raise the value of all remaining shares in circulation.

To understand this better, let's consider a scenario with a fixed market capitalization for Ethereum. In such a case, the market capitalization should ideally be independent of the number of coins in circulation and dictated by factors like the demand for the currency and speculation about its future market cap.

Now, imagine cutting the supply of circulating coins in half while keeping all other factors constant. With a fixed market capitalization, each remaining coin would have to be priced twice as much to compensate for the halved supply. This intuition illustrates why burning base fee revenues benefits the remaining holders of the native currency. As coins are taken out of circulation, the value of the remaining coins should increase, potentially leading to price appreciation.

Effect on Inflation Rate

As mentioned in the text, the burning of base fee revenues has the aggregate effect of reducing the inflation rate of Ethereum. Prior to EIP-1559, Ethereum's inflation rate was roughly four percent. However, with the implementation of the new transaction fee mechanism, the inflation rate has decreased to approximately 2%.

This reduced inflation rate benefits all holders of the native currency, as the pace of coin issuance is slowed down. By not printing coins as rapidly as before, the switch to the transaction fee mechanism should be favorable to all holders of Ethereum, allowing them to benefit from potentially increased coin value.

Impact on Miners' Revenue

While the burning of base fee revenues does result in a loss of revenue for miners, the consequences may not be as catastrophic as initially perceived. In the post-EIP-1559 world, miners still have multiple revenue streams at their disposal:

1. **Block Rewards**: Miners receive two new ethers for every block they successfully mine. This block reward serves as a steady income stream for miners.
2. **Excess Bid Fees**: Users have the option to bid above the base fee for their transactions. Any amount exceeding the base fee is transferred to the miner. Therefore, miners can still earn additional revenue through these excess bids in a post-EIP-1559 world.
3. **Special Treatment Transactions**: Certain transactions may require special treatment, such as being included as the first transaction in a block. Users may offer higher fees to incentivize miners to prioritize their transactions. This can result in a significant source of additional revenue for miners.
4. **Miner Extractable Value (MEV)**: Miner Extractable Value is a unique source of revenue for miners that involves inspecting the content of transactions and potentially optimizing block inclusion to their advantage. This can be especially significant in decentralized finance (DeFi) transactions, where miners can profit from various opportunities.

Despite the loss of base fee revenues, miners retain multiple avenues to generate income, which can soften the impact of the transition to the new fee mechanism.

In conclusion, while the burning of base fee revenues does have consequences for miners and holders of the native currency, it may not be as detrimental as initially anticipated. The reduced inflation rate benefits all holders of the currency, potentially leading to increased coin value. Additionally, miners can still rely on block rewards, excess bid fees, special treatment transactions, and Miner Extractable Value to maintain revenue in the post-EIP-1559 world. Overall, the Ethereum community successfully navigated the implementation of this major update, showcasing the decentralized decision-making process and resilience of the network.

11.10.2 Impact on Miners and Blockchain Security

The implementation of EIP-1559, specifically the burning of base fee revenues, has significant implications for miners and the overall security of the Ethereum blockchain.

Decreased Miner Revenue

With the introduction of the transaction fee mechanism in EIP-1559, miners experience a decrease in their revenue. While the base fee revenues are no longer directly passed on to miners, they still receive block rewards, excess bid fees, and revenue from special treatment transactions and Miner Extractable Value (MEV). However, the reduction in base fee revenues impacts the overall miner profitability.

To understand this impact better, let's consider an example where a miner is barely covering their operating costs with their previous revenue stream. The decrease in revenue due to the burned base fee revenues might result in a situation where their income is significantly lower, say around 20-30%. In such a scenario, the miner may choose to stop mining altogether to avoid losses. However, it is important to note that the decrease in revenue is not as catastrophic as it may initially seem. Some miners might continue to find mining profitable due to the other revenue streams, and the decrease in hash rate can create a feedback loop that affects mining difficulty and rewards.

Feedback Loop on Mining Difficulty

As some miners exit the network due to decreased profitability, the overall hash rate of the Ethereum network decreases. The difficulty adjustment algorithm in proof-of-work blockchains, like Ethereum, ensures that blocks are produced at a consistent rate (e.g., one block every 13 seconds).

Suppose a miner chooses to remain in the network despite the decreased revenue. Since the hash rate has reduced, the cryptographic puzzles become easier to solve. As a result, each hash attempt is more likely to yield a solution, leading to more frequent block mining for the remaining miners.
This feedback loop tends to stabilize the network to a new equilibrium where the remaining miners find it more profitable to mine despite the decrease in base fee revenues. The adjustment in mining difficulty maintains the block production rate, which is crucial for the proper functioning of the Ethereum blockchain.

Blockchain Security Concerns

One potential concern arising from the decrease in hash rate is the impact on the security of the Ethereum blockchain. A lower hash rate makes the network more susceptible to 51% attacks, where an attacker gains control of a majority of the network's computing power.
A successful 51% attack would allow the attacker to manipulate transactions, double-spend coins, and disrupt the network's integrity. The burning of base fee revenues may lead to some miners exiting the network, making it easier for potential attackers to acquire the necessary hash rate for a 51% attack. However, it is worth noting that this concern was not a significant sticking point during the discussions around EIP-1559 in the Ethereum community. Many leaders in the community believed that the remaining block rewards would provide sufficient incentives for miners to maintain network security.

In retrospect, the Ethereum network's security has not been severely compromised following the implementation of EIP-1559. While the decrease in hash rate can be a potential concern, the network has continued to operate securely with miners adapting to the new economic model.

Overall, the impact of EIP-1559 on miners and blockchain security was a major point of discussion in the Ethereum community. Despite the decrease in miner revenue, the network has demonstrated resilience, and the feedback loop on mining difficulty has contributed to maintaining network security.

The implementation of EIP-1559 sparked extensive discussions within the Ethereum community, with varying perspectives on its impact on miners and overall network security. Despite the decrease in miner revenue, the community managed to successfully deploy this major update in a decentralized and distributed manner. In the next chapter, we will explore proof-of-stake blockchains as an alternative to proof-of-work, focusing on their significance and benefits.