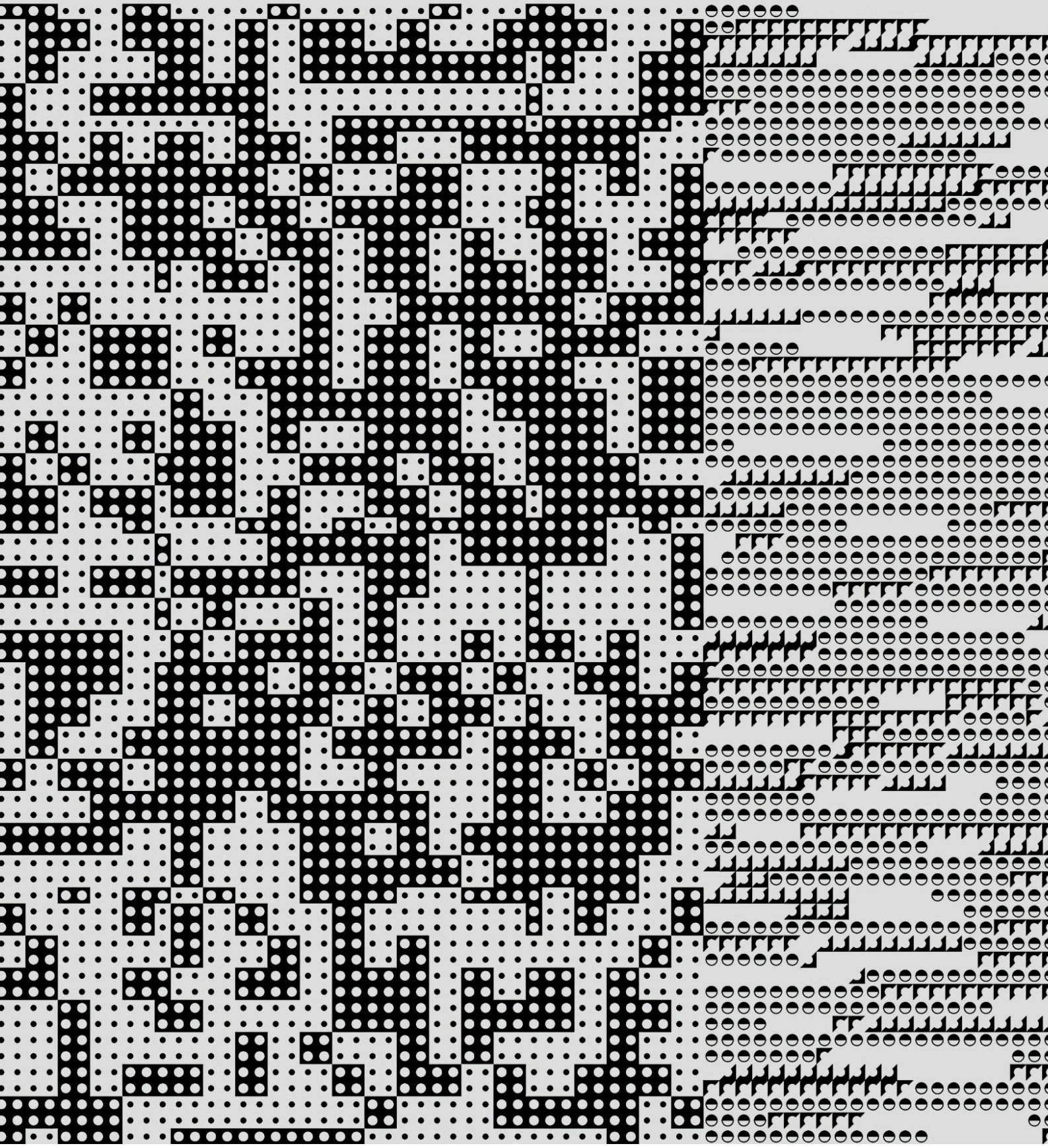

Blockchain

Roberto Vicario / 2024-2025

UNIVERSITAS STUDIORUM INSUBRIAE

Contents

1 Blockchain **3**
1.1 State Machine Replication (SMR) . . . 3
1.2 Blockchain Stack . . . 4

2 Consensus Problem **4**
2.1 Consensus Algorithm . . . 5
2.2 Permissioned vs. Permissionless . . . 5

3 Network Models **5**
3.1 Synchronous . . . 5
3.2 Asynchronous . . . 6
3.3 Partial Synchronous . . . 6

4 Byzantine Fault Tolerance (BFT) Problem **7**
4.1 Fault Tolerance . . . 7

5 Byzantine Broadcast (BB) **7**
5.1 Dolev-Strong Protocol . . . 8

6 Byzantine Agreement (BA) **9**
6.1 FLP (Fischer-Lynch-Patterson) Impossibility Result . . . 9

7 Rotating Leaders **10**
7.1 DLS (Dwork-Lynch-Stockmeyer) Impossibility Result . . . 11
7.2 Tendermint Protocol . . . 12

8 Longest Chain (LC) **12**
8.1 Sybil Attacks . . . 13
8.1.1 Proof of Stake (PoS) . . . 13
8.1.2 Proof of Work (PoW) . . . 14

9 Nakamoto Protocol **14**

10 Proof of Work (PoW) **15**
10.1 Incentives . . . 15
10.2 Transaction Fees . . . 16

11 Proof of Stake (PoS) **16**
11.1 Slashing . . . 16

12 Execution Layer **17**

13 UTXO (Unspent Transaction Output) Model **17**

13.1 Merkle Tree 17

14 Account-Based Model **18**

1 Blockchain

The *Blockchain* is a registry technology that allows to record information in an immutable, transparent and secure way without the need for a central authority. It is called "blockchain" (chain of blocks) because the data is grouped into blocks, linked together in chronological order through cryptography.

> **Decentralization:** Control and decision-making aren't held by a central authority, like a company or government. Instead, power is distributed across a network of independent participants. Data is shared among many nodes (computers) distributed in a peer-to-peer network.

Key components:

- **Transaction:** A record of an action or event, like sending money.
- **Block:** A container for a list of transactions. Each block contains a unique identifier (hash) and a reference to the previous block, forming a chain. The first block in the chain is called the *Genesis Block*.
- **Node:** A computer that participates in the blockchain network, maintaining a copy of the ledger and validating transactions.

1.1 State Machine Replication (SMR)

A blockchain is an extension of the *State Machine Replication (SMR)* model. In this model, a distributed system maintains a consistent state across multiple nodes. Each node processes transactions in the same order, ensuring that all nodes reach the same state.

$\boxed{DEFINITION}$

Let S be a set of states, T be a set of transactions, and f be the function that maps a state and a transaction to a new state.

Assumption: If all replicas receive the same sequence of transactions $t_1, t_2, \dots, t_n$, they will all reach the same final state s_n and produce the same output.

The state machine replication model can be defined as:

$$S = \{s_0, s_1, s_2, \dots, s_n\}$$

$$T = \{t_0, t_1, t_2, \dots, t_m\}$$

$$f : S \times T \to S$$

Where:

- s_0 is the initial state of the system.
- s_i is the state of the system after processing transaction t_i.
- $f(s_i, t_i) = s_{i+1}$ is the function that updates the state of the system after processing transaction t_i.

1.2 Blockchain Stack

The main layers of the blockchain stack are:

1. **Network Layer:** The underlying infrastructure that connects all nodes in the blockchain network. It ensures that data can be transmitted securely and efficiently between nodes.
2. **Consensus Layer:** The protocol that allows nodes to agree on the state of the blockchain. It ensures that all nodes have a consistent view of the ledger, even in the presence of faulty or malicious nodes.
3. **Data Layer:** The actual data stored in the blockchain, including transactions and smart contracts. This layer is responsible for maintaining the integrity and immutability of the data.
4. **Execution Layer:** The user-facing applications and interfaces that interact with the blockchain. This layer includes wallets, decentralized applications (dApps), and other tools that allow users to interact with the blockchain.

2 Consensus Problem

In decentralized systems, no single person or computer is in charge, so how do all the independent nodes agree on what's true? That's the *Consensus Problem*:

> **Problem:** *How can a group of participants, who don't fully trust each other, agree on a single version of the truth?*

2.1 Consensus Algorithm

A *Consensus Algorithm* is a mechanism used in distributed systems to achieve agreement on a single data value among distributed processes or systems. It is essential for ensuring that all nodes in a blockchain network have a consistent view of the ledger. Consensus algorithms are designed to handle various types of failures, including network partitions, node crashes, and malicious behavior.

$\boxed{ASSUMPTIONS}$

An algorithm reaches consensus under the following assumptions:

- **Safety:** The system guarantees that all honest nodes will agree on the same value, even in the presence of faulty nodes.
- **Liveness:** The system guarantees that a decision will eventually be reached, provided that a sufficient number of honest nodes are present.

2.2 Permissioned vs. Permissionless

In a blockchain network, participants can be classified into two categories based on their access rights:

- **Permissioned:** Only authorized participants can join the network and validate transactions. This type of network is often used in private blockchains or consortium blockchains, where a group of organizations collaborates.
- **Permissionless:** Anyone can join the network and participate in the consensus process. This type of network is typically used in public blockchains, where anyone can become a node and validate transactions.

3 Network Models

A blockchain network can be implemented in various ways, depending on the requirements and goals of the system. The implementation can be classified into three categories based on the communication model used by the nodes:

3.1 Synchronous

In a *Synchronous* network, all nodes are assumed to have synchronized clocks and can communicate with each other within a known time frame. This means that messages sent between nodes will arrive within a predictable time limit.

ASSUMPTIONS

This model assumes that:

- **Timing:** All nodes have synchronized clocks, meaning they can agree on the current time.
- **Message Delivery:** Messages sent between nodes will arrive within a known time frame.
- **Processing Time:** The time taken to process a message is known and bounded.

> **Consideration:** These networks are easier to reason about, but they are less common in real-world scenarios due to network latency and clock synchronization issues.

3.2 Asynchronous

In an *Asynchronous* network, there are no assumptions about the timing of message delivery or the processing time of messages. Nodes can communicate with each other, but there is no guarantee that messages will arrive within a specific time frame. Assumptions are mainly the opposite of synchronous networks, the service could be delayed or unavailable.

> **Consideration:** These networks are more realistic for real-world scenarios, but they are harder to reason about due to the lack of timing guarantees. They have a higher risk of network partitions and message loss, and theoretical limits on consensus.

3.3 Partial Synchronous

In a *Partial Synchronous* network, there are some guarantees about message delivery and processing times, but these guarantees may not hold at all times. This model is a compromise between synchronous and asynchronous networks.

ASSUMPTIONS

This model assumes that:

- **Timing:** Nodes may not have synchronized clocks, but the network can eventually becomes stable enough for synchronously.
- **Message Delivery:** Messages sent between nodes will eventually arrive, but there may be periods of time when messages are delayed or lost.

- **Processing Time:** There exists a bound on the time taken to process a message, but this bound is unknown or does not hold at all times.

Consideration: These networks combine real-world practices with theoretical guarantees to reach consensus. It assumes that even if the messages are delayed, they will eventually be delivered.

4 Byzantine Fault Tolerance (BFT) Problem

In a distributed network, not every participant can be trusted. Some may fail, lie, or act maliciously. This leads to the *Byzantine Fault Tolerance Problem*:

Problem: *If all the generals coordinate their attack and strike at the same time, the battle is won. However, if two generals falsely claim they will attack but instead retreat, the battle is lost. This failure in coordination undermines the entire strategy.*

4.1 Fault Tolerance

Fault tolerance is the ability of a system to continue operating correctly even when some of its components fail:

- **Crash Faults:** Nodes may crash or become unresponsive.
- **Byzantine Faults:** Nodes may behave arbitrarily, including sending conflicting information to different nodes.

> **Theorem:** Let n be the total number of nodes in a distributed system and f be the maximum number of faulty nodes. To guarantee that honest nodes can reach consensus despite the presence of these faulty nodes, it is necessary that:

$$f \leq \frac{n-1}{3} \tag{1}$$

5 Byzantine Broadcast (BB)

> ***Synchronous***

This method allows a source node to broadcast a message to all other nodes in the network, ensuring that even if some nodes are faulty, the remaining honest nodes can still reach consensus on the message.

$\boxed{ALGORITHM}$

1. **Initialization:** The source node sends the message to all nodes.
2. **Propagation:** Each node forwards the message received from the source node to all other nodes.
3. **Acknowledgment:** Each node applies a decision rule (e.g., majority voting) based on the messages received to agree on the final value.
4. **Termination:** The protocol terminates when all honest nodes agree on the same value.

Single Point of Failure: This protocol relies on a single source node to initiate the broadcast. If this node is compromised, it can send conflicting messages to different nodes, potentially disrupting consensus and causing the entire system to never reach consensus.

5.1 Dolev-Strong Protocol

> ***Synchronous***

Dolev-Strong protocol allows a source node to broadcast a message to all other nodes in the network, ensuring that even if some nodes are faulty, the remaining honest nodes can still reach consensus on the message.

$\boxed{ALGORITHM}$

1. **Initialization:** The sender sends a signed message m to each node:

$$M_0 = (m, \sigma_S(m))$$

2. **Propagation:** For rounds $r = 1, \dots, f$, each node forwards the message to all other nodes, including its own signature i:

$$M_{r-1} = (m, \sigma_{i_0}(m), \sigma_{i_1}(m), \dots, \sigma_{i_{r-1}}(m))$$

3. **Acknowledgment:** Each node collects all received messages M_r. Messages with invalid values in the chain are discarded.
4. **Termination:** The consensus is reached when nodes reach the quorum of $2f+1$ messages. The final value is the one with the highest number of votes.

> **FLM (Fischer-Lynch-Merritt) Impossibility Result:** In synchronous systems, the absence of a *Public Key Infrastructure (PKI)*, makes impossible to achieve consensus.

6 Byzantine Agreement (BA)

> ***Asynchronous***

This method does not rely on a single source node to initiate the broadcast. Instead, it allows all nodes to participate in the consensus process, ensuring that even if some nodes are faulty, the remaining honest nodes can still reach consensus on the message.

$\boxed{ALGORITHM}$

1. **Initialization:** Each node starts with an initial private value.
2. **Agreement:** All honest nodes decide the same value.
3. **Validity:** If all honest nodes start with the same value, they will all decide on that value.
4. **Termination:** The protocol terminates when all honest nodes agree on the same value.

6.1 FLP (Fischer-Lynch-Patterson) Impossibility Result

> ***Asynchronous***

In asynchronous systems, does not exist a deterministic algorithm that can guarantee consensus in the presence of even a single faulty node. This could lead to a situation where the system never reaches consensus, as nodes may wait indefinitely for messages from other nodes.

$\boxed{PROOF}$

Let each node hold a private input value of either 0 or 1. Define a family of initial configurations X_i, where the first i nodes have value 0 and the remaining $n-i$ nodes have value 1:

$$X_i = (\underbrace{0, 0, \dots, 0}_{i}, \underbrace{1, 1, \dots, 1}_{n-i})$$

Since $X_0 = (0, 0, \dots, 0)$, by *Validity*, the output must be 0, and is a 0-configuration. Similarly, $X_n = (1, 1, \dots, 1)$ must lead to output 1, and is a 1-configuration.

If we gradually flip the input bits from 0 to 1, moving from X_0 to X_n, there must exist a first index i such that:

- X_{i-1} is still a 0-configuration, but X_i is not a 0-configuration, and is not yet a 1-configuration either.

> **Lemma 1:** There exists at least one *Ambiguous Configuration*, where the output is not determined and *Termination* is not guaranteed.

Let C be an ambiguous configuration, one from which both decision values 0 and 1 are still possible through different executions. Then, there exists at least one event e (e.g., the delivery of a message to a process) such that executing e from C leads to another ambiguous configuration.

> **Lemma 2:** Even if an event is applied to an ambiguous configuration, it does not necessarily break the bivalence: the system can remain in a state where both outcomes (0 or 1) are still possible. This prevents the system from "forcing" a definitive decision in a finite number of steps.

Using Lemma 1 (existence of an ambiguous configuration) and Lemma 2 (persistence of ambiguity), one can construct an infinite, fair execution in which the system never reaches a decision, violating the *Termination* property of consensus.

Conclusion: Therefore, deterministic consensus is impossible in an asynchronous system with even one possible faulty node.

7 Rotating Leaders

> ***Partially Synchronous***

This method proposes a solution for *Single Point of Failure* problems by allowing nodes to take turns being the leader, ensuring that no single node has control over the entire system.

$\boxed{ALGORITHM}$

1. **Initialization:** A leader is selected.
2. **Proposal:** The leader proposes a value to the other nodes.
3. **Voting:** Each node votes on the proposed value, and if a majority agrees, the value is committed.
4. **Rotation:** After a round, the leadership role rotates to the next node.

7.1 DLS (Dwork-Lynch-Stockmeyer) Impossibility Result

> ***Partially Synchronous***

In a partially synchronous system, where message delivery time becomes bounded only after some unknown *Global Stabilization Time (GST)*, no deterministic algorithm can guarantee both safety and liveness in the presence of even one faulty process.

$\boxed{PROOF}$

1. Let N_2 be the faulty process, with N_1 and N_3 honest. Suppose N_1 has input 1 and N_3 has input 0.
2. The faulty process N_2 behaves inconsistently, sending messages to N_1 claiming its input is 1, and to N_3 claiming its input is 0.
3. Meanwhile, the adversary delays all communication between N_1 and N_3 until after both have made their decisions.
4. Because GST is unknown and message delays before GST can be unbounded, N_1 and N_3 cannot distinguish between delayed messages and crashed processes. Consequently, N_1, believing both it and N_2 are honest, may decide on 1; similarly, N_3, also trusting N_2, may decide on 0. This leads to:

   $$v_1 \neq v_3$$

   Where v_1 and v_3 are the decisions made by N_1 and N_3, respectively.

Conclusion: Therefore, no deterministic algorithm can achieve consensus in a partially synchronous system with even one faulty process.

> **CAP Principle:** No distributed system can simultaneously guarantee *Consistency*, *Availability*, and *Partition Tolerance*.

7.2 Tendermint Protocol

> ***Partially Synchronous***

Tendermint is a deterministic consensus protocol designed to work in the partially synchronous model, tolerating faulty nodes giving up safety during asynchronous periods.

$\boxed{ALGORITHM}$

1. **Propose:** A designated proposer (deterministically selected from the validator set) broadcasts a proposed block B_h for height h to all validators.
2. **Prevote:** Upon receiving a valid proposal B_h, each validator v_i broadcasts a prevote message:

$$\text{prevote}_i(h, r) = \begin{cases} \text{hash}(B_h), & \text{if } B_h \text{ is valid and received} \\ \text{none}, & \text{otherwise} \end{cases}$$

   Where r is the current round number.
3. **Precommit:** If a validator observes prevotes for the same block hash from at least $\frac{2n}{3}$ validators, it broadcasts a precommit message:

$$\text{precommit}_i(h, r) = \begin{cases} \text{hash}(B_h), & \text{if prevotes}(h, r) \geq \frac{2n}{3} \text{ for } B_h \\ \text{none}, & \text{otherwise} \end{cases}$$

4. **Commit:** If a validator observes precommits for the same block hash from at least $\frac{2n}{3}$ validators, it commits B_h as the block at height h, finalizes the round, and proceeds to height $h + 1$.

8 Longest Chain (LC)

Longest Chain is a data structure used in blockchain systems to maintain a single, agreed-upon history of transactions. This structure is designed to guarantee that all nodes could participate in the consensus process, even when they do not know the genesis block.

Problem: *Nodes may disagree on the current state because they have seen different versions of the chain. Is there a way to randomly sample the leader from an unknown set of participants?*

$\boxed{ASSUMPTIONS}$

An algorithm reaches consensus under the following assumptions:

1. **Unknown Genesis Block:** No node knows the genesis block, before the start of the consensus process.
2. **Leader Verification:** It is easy for all nodes to verify whether a given node is the leader.
3. **Leader Selection:** No node can influence the probability of being selected as the leader.
4. **Predecessor Requirement:** For each new proposal, must exists a predecessor block in the chain from the previous round.
5. **Chain Status:** At all times, all correct nodes know the same set of blocks and their predecessors.

8.1 Sybil Attacks

A *Sybil Attack* occurs when a single entity creates multiple identities to gain disproportionate influence in a network.

> **Theorem:** Given a network of n nodes, each with a distinct hashrate $\mu_1, \mu_2, \dots, \mu_n$. In each round of leader selection, the probability that node i is chosen as the leader is proportional to its hashrate and is given by:

$$\frac{\mu_i}{\sum_{j=1}^{n} \mu_j}$$

To prevent Sybil attacks, the system must ensure that creating multiple identities is costly or requires a significant investment of resources. For this reason, were implemented two main mechanisms: *Proof of Work (PoW)* and *Proof of Stake (PoS)*.

8.1.1 Proof of Stake (PoS)

The chance of being chosen to propose or validate a block generally depends on the amount committed. This approach helps limit the influence of any single participant and discourages the creation of many identities. PoS can be integrated into consensus mechanisms in these ways:

- **PoS + BFT:** The quorum is easily achieved by selecting the nodes with the highest stake.
- **PoS + LC:** The longest chain selects the leader by the depth of the chain, which is proportional to the stake held by the nodes.

8.1.2 Proof of Work (PoW)

In this mechanism, the nodes called miners, compete to solve a cryptographic *Hard Puzzle* by finding a nonce that, when combined with the block's data and hashed, produces a hash value below a specified target:

$\boxed{PROOF}$

Given a cryptographic hash function H, find an input x such that:

$$H(x) \leq \tau$$

Where τ is the current difficulty target.

Since H behaves like a random function, the only viable strategy is brute-force search. The difficulty of the puzzle is adjusted by changing τ:

- Smaller $\tau \Rightarrow$ fewer hash outputs satisfy the condition $\Rightarrow$ harder puzzle
- Larger $\tau \Rightarrow$ more outputs satisfy the condition $\Rightarrow$ easier puzzle

PoW can be integrated into consensus mechanisms in these ways:

- **PoW + BFT:** Integrating PoW with BFT consensus can introduce instability. Fluctuations in the network's total computational power may disrupt predictable leader selection, undermining the reliability and security guarantees of BFT protocols. As a result, combining PoW with BFT is generally discouraged.
- **PoW + LC:** Nodes compete to solve computational puzzles, and the chain with the most accumulated proof of work is considered the valid one. This combination forms the basis of *Nakamoto Protocol*.

9 Nakamoto Protocol

> **Partially Synchronous**

Nakamoto Protocol is a consensus mechanism designed for PoW-based blockchains. It elects a leader in each round based on the computational effort expended by nodes, allowing them to agree on a single chain of blocks.

$\boxed{ALGORITHM}$

1. **Puzzle Solving:** Each node attempts to solve a cryptographic puzzle.
2. **Leader Election:** The first node to find such x becomes the leader and broadcasts the block as its proposal for the next block in the chain.
3. **Chain Extension:** Honest nodes always extend the chain with the highest total work, the one requiring the most cumulative PoW effort.
4. **Difficulty Adjustment:** The protocol adjusts τ over time to maintain a stable block production rate and reduce accidental forks. A typical target is one block every fixed time interval.

10 Proof of Work (PoW)

Proof of Work (PoW) is a consensus mechanism used in blockchain systems to ensure that all participants agree on the state of the ledger. It requires participants (miners) to solve complex mathematical problems in order to add new blocks to the blockchain.

10.1 Incentives

With the purpose to create a secure and decentralized network, PoW mechanisms provide economic incentives for miners to participate honestly. The main incentives include:

- **Block Rewards:** Miners receive a reward for successfully mining a block, which is typically a fixed amount of cryptocurrency.
- **Transaction Fees:** Miners can also collect fees from transactions included in the blocks they mine.

Selfish Mining: A node can deviate from the protocol to increase its own rewards, potentially leading to a situation where it can earn more than its fair share of rewards.

There are two main cases to consider regarding selfish mining:

- **Case with $\alpha > 0.5$:** Node A can orphan honest blocks and earn approximately 100% of the rewards.
- **Case with $\alpha < 0.5$:** It is still possible to earn more than α of the total rewards through a strategy that: delays block announcements, selectively orphans blocks, and exploits tie-breaking in an adversarial manner.

In general, a selfish miner can obtain a share of rewards greater than their relative mining power α if α exceeds approximately *0.33*. This means that even with less than one-third of the total mining power, a selfish miner can outperform honest mining by strategically withholding and releasing blocks.

10.2 Transaction Fees

Transaction fees are an essential part of PoW mechanisms, as they provide an additional incentive for miners to include transactions in the blocks they mine. The fees are typically paid by users who want their transactions to be processed quickly.

Block Size Problem: The limited capacity of blocks leads to competition among transactions for inclusion, resulting in the need for transaction fees.

11 Proof of Stake (PoS)

Proof of Stake (PoS) is a consensus mechanism used in blockchain systems as an alternative to Proof of Work (PoW). It aims to achieve consensus without requiring extensive computational resources, making it more energy-efficient and environmentally friendly.

11.1 Slashing

Just like in PoW, validators in PoS systems are incentivized to act honestly through rewards and penalties. To discourage malicious behavior, PoS protocols often implement a mechanism called *Slashing*. This involves the partial or full loss of staked funds for validators who engage in misbehavior, such as:

- **Double Signing:** Attempting to validate multiple blocks at the same height.
- **Inactivity:** Failing to participate in the consensus process for an extended period.
- **Attestation Failure:** Not voting on blocks or not following the protocol rules.

12 Execution Layer

The *Execution Layer* is a critical component of blockchain systems, responsible for executing transactions and maintaining the state of the blockchain. It operates on top of the consensus layer, which ensures that all nodes agree on the order of transactions.

13 UTXO (Unspent Transaction Output) Model

The *UTXO* model is used in Bitcoin. It represents the state of the blockchain as a set of unspent transaction outputs, which can be used as inputs for new transactions.

$\boxed{DEFINITION}$

Given a UTXO block U_i in the blockchain, and let $TX_i = \{tx_1, ..., tx_n\}$ be the set of transactions included in the block, organized in a Merkle tree with root hash h_{root}. The block U_i can be defined as:

$$U_i = (H_{i-1}, MT(h_{\text{root}})) \tag{2}$$

13.1 Merkle Tree

A *Merkle Tree* is a data structure that allows efficient verification that a transaction is included in a block, without needing all the data.

$\boxed{SEARCHING}$

1. **Initialization:** Client stores the root hash of the Merkle tree h_{root}.
2. **Forwarding:** Prover sends to the client the transaction hash TX_i and the Merkle proof:

   $$\Pi = \{h_1, h_2, ..., h_k\}$$

   Where each h_i is a hash at some level of the tree.
3. **Verification:** Starting from tx_i, the client combines it with each h_i, following the tree path:

   $$h' = H(H(H(tx_i \parallel h_1) \parallel h_2) \parallel ... \parallel h_k)$$

4. **Termination:** The client accepts if the final hash equals the stored Merkle root:

$$h' = h_{\text{root}}$$

14 Account-Based Model

The *Account-Based Model* is used in Ethereum. It represents the state of the blockchain as a set of accounts, each with its own metadata.

The two main types of accounts in the account-based model are:

- **Externally Owned Account (EOA):** Controlled by a private key, can send transactions.
- **Contract Account:** Contains code and storage, can execute smart contracts.