

# MAIN PDF

## Chapter 1: Introduction

Unlike Client-Server architectures, where there are end-hosts (clients) and special hosts (servers), in Peer-to-Peer (P2P) systems, there are only end-nodes (peers) that talk directly to each other. These peers can go online or offline at any time ("on/off" behavior), and the system must deal with this frequent change (called churn).

However, even in P2P systems, servers are still used — but only as **bootstrap servers**. These servers help new peers join the P2P network more easily.

Connections between peers in a P2P system are called **transient**, meaning peers connect and disconnect from the network very often.

Also, because peers might get a new IP address every time they join the network, we cannot use IP addresses to find resources. So, a different method at the **application layer** is used instead.

### Definition 1.1 (P2P System)

A peer-to-peer system is a group of independent entities (called peers) that can organize themselves and share distributed resources over a computer network. The system uses these resources to provide a service in a fully or partly **decentralized** way.

### P2P Systems

#### Semi-Decentralized Systems

In Semi-Decentralized systems, a **central server** is used only for **helping peers find each other** (node discovery). The **actual file sharing** happens directly between peers.

A good example is **Napster**, which was launched in 2001. Napster used **servers only to help users find** other peers who had the file they wanted. The actual **file transfer** happened directly between peers, so only a **few servers were needed**.

For the first time, users were called **peers**, and systems like this were called **peer-to-peer systems**.

Napster had many **advantages** common to P2P systems — such as allowing peers to act as **both client and server**.

But it also had **weaknesses**, because it still depended on a **central server** for finding other peers (called **node discovery**).

This central server was a **bottleneck** in the design, and also became a target of **legal actions**.

### Fully Decentralized Systems

In Fully Decentralized systems, there is **no central server at all**. Peers are completely equal and use **direct connections** to **search and share** resources.

**Gnutella** is similar to Napster, but it does **not use any central server**.

Main problems with Gnutella:

1. **High network traffic**
2. **No organized (structured) search**
3. **Free-riding** (some users download files but don't share anything)

### P2P Overlay Network

In P2P systems, there is an **overlay network** that works at the **application level**, built on top of the regular **IP network**(the Internet).

A **P2P protocol**, which runs on this overlay network, defines the **messages** that peers send to each other.

### Unstructured Overlay

There are two main problems in unstructured P2P networks:

- ♦ **How to join (bootstrap) the network?**
- ♦ **How to find content without using a central index?**

Some **search (lookup) algorithms** can be used to solve these problems, but they are usually **not scalable** and have **high performance costs**:

- **Flooding:** Flooding is a search method where a peer sends a request to **all its neighbors**, and then those neighbors forward it to **all their neighbors**, and so on — until the content is found or a limit is reached.
- **Expanding Ring:** is an improved version of Flooding. The search is done in **steps**, starting with a small **Time-To-Live (TTL)** value. If the file isn't found, the TTL is increased and the search is repeated with a **wider range**.
- **Random Walk:** In Random Walk, the search request is sent to **one random neighbor**, which forwards it to **another random neighbor**, and so on — like taking steps randomly through the network.

## Structured Overlays

In **structured overlay networks**, peers choose their **neighbors** based on specific **rules**. This creates a well-organized (structured) network.

The main goal is to support **scalability** by offering:

- **Key-based lookup** (you can find data using a key)
- **Search complexity** that is predictable, for example  **$O(\log N)$**

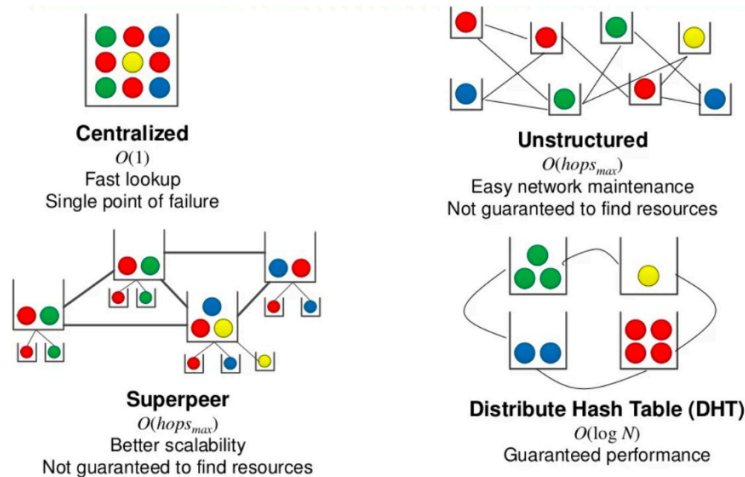
Some examples of structured overlay systems are **DHT-based (Distributed Hash Table)** solutions like **Chord**, **Pastry**, and **Kademlia**.

## Hierarchical Overlays

In this type of network, **peers connect to Super-Peers**, which store information (or "index") about the resources of normal peers.

Only the **Super-Peers** send search requests (**flooding**), but files are still shared directly between regular peers.

This method improves **scalability** and **reduces lookup time**, but if **Super-Peers leave the network** (called **churn**), it can cause problems more easily.



## ✓ Chapter 2: Distributed Hash Tables

The main idea is to **split the hash table into parts** and **distribute them across several servers**. The system uses the **hash of a resource** as key, and this key is used to map the resource to one server (web cache).

Each machine (or user) can **locally calculate** which web cache should have the resource, just by using the key. This idea is used in **DHT (Distributed Hash Table)** systems for P2P networks.

However, there is a problem in **dynamic systems**, where the number of servers changes. If the hash method depends directly on how many servers there are, then when a server is added or removed, about **99% of the keys need to be moved**. This causes **many messages to be exchanged**, which is inefficient.

**Consistent Hashing** is a set of hash methods that **solve this problem**. It makes sure that when **nodes are added or removed**, only a **small number of keys** need to be moved.

In this method, each node is responsible for an **interval of hash values** (instead of random keys). When a node joins or leaves, these intervals are **split or merged**, and only **neighboring nodes** exchange keys.

### Peers Joining and Leaving

When a **new node joins**, the keys between the new node and the previous node in the hash ring are moved to the new node. These keys are no longer linked to their old nodes.

When a **node leaves** the hash ring, only the keys that belonged to that node are rehashed and moved. We do not need to remap all the keys.

If a node **suddenly disconnects**, all the data stored on it will be lost unless the data is also saved on other nodes. To prevent this problem, we can:

- ◊ add **redundancy** by copying data to multiple nodes (data replication)
- ◊ perform **periodic refresh** to keep information updated and avoid loss

When the hash table changes size, on average only  $k/n$  keys need to be remapped, where **k** is the total number of keys and **n** is the number of servers (nodes).

## Chapter 3: Kademlia

Kademlia is a protocol used by some of the biggest public DHTs, such as:

- BitTorrent Mainline DHT
- Ethereum P2P network
- IPFS

Kademlia has three important features that other DHTs don't have:

1. **Routing information spreads automatically** during lookups. The list of known nodes is updated with the most recently contacted ones.
2. **Parallel routing**: Kademlia can send multiple requests at the same time to different nodes in the same group (called a k-bucket), which makes lookups faster by avoiding delays from waiting for responses.
3. **Iterative routing**: At every step of the search, the node you ask sends a response back to the original requester—even if it can't answer the query directly.

### Structure:

Kademlia uses the **leaves of a complete binary Trie** to create its logical identifier space. However, **not every leaf represents a node (peer)**.

For example, in Figure 3.1, only the **leaves with circles** around them are actual nodes.

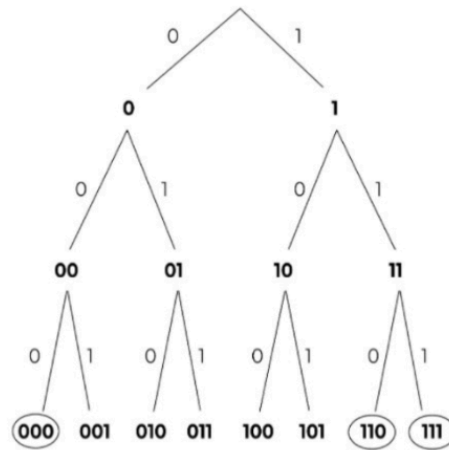


Figure 3.1: Trie

The way keys (content) are divided among nodes must follow the rules of **consistent hashing**.

**Definition 3.1 (Partitioning rule):** A key is assigned to the node that shares the **longest common prefix** with the key.

In other words, find the node whose ID has the longest matching start (prefix) with the key, and assign that key to that node.

### How do we calculate the distance between two nodes?

Kademlia uses the **XOR metric** to do this.

It calculates the **XOR** (exclusive OR) between a key and a node's ID, then treats the result as a number to find the distance.

This method:

- Guarantees there is **only one closest node** to any key

- Ensures the distance between two nodes is the **same in both directions** (distance from A to B is the same as from B to A).

### Routing Table

To find data, Kademlia's main idea is to **store a small number of node IDs and their IP addresses**—about **logarithmic in the total number of nodes**. These nodes are chosen based on their position in the identifier trie.

### Key Lookup

The idea for looking up a key is:

1. Find the node in your routing table that is **closest to the key**, using the XOR distance to decide if there is a tie.
2. While the closest node you know doesn't have the key and hasn't replied yet:
  - i. Ask that closest node for the key or for a node that is even closer to the key.
  - ii. If the node replies with a closer node, update your list of closest nodes (called the current bucket).

At each step, the XOR distance gets about **half as big**, and your list of possible nodes (k-buckets) becomes smaller.

## Chapter 4: BitTorrent

The goal of CDNs (Content Distribution Networks) is to **deliver web content** (like videos or files) to **hundreds of thousands or even millions of users** at the same time. This is done by **copying the data or service** across multiple "mirror" servers.

In a **P2P CDN**, the **first request** for a file is handled by a **central server**. After that, other users (called **peers**) who already downloaded the file (**seeders**) can share it directly with others, so the original server is no longer needed.

### Protocol Overview (BitTorrent)

1. Seeder: The **.torrent file** is uploaded to a **Torrent Server**, and it includes the **Tracker address** and the **infohash** (unique identifier) of the file.

2. Seeder: **connects to the Tracker** and tells it: "I'm online and I have the file." At this point, it's the only peer with the file.
3. Another peer **downloads the .torrent file** and opens it with a **BitTorrent client**.
4. This peer also **contacts the Tracker**, tells it "I'm here", and gets a **list of peers (called the swarm)** that are sharing the same file.  
(The tracker address is inside the .torrent file.)
5. The peer then **connects to other peers in the swarm** to start downloading.

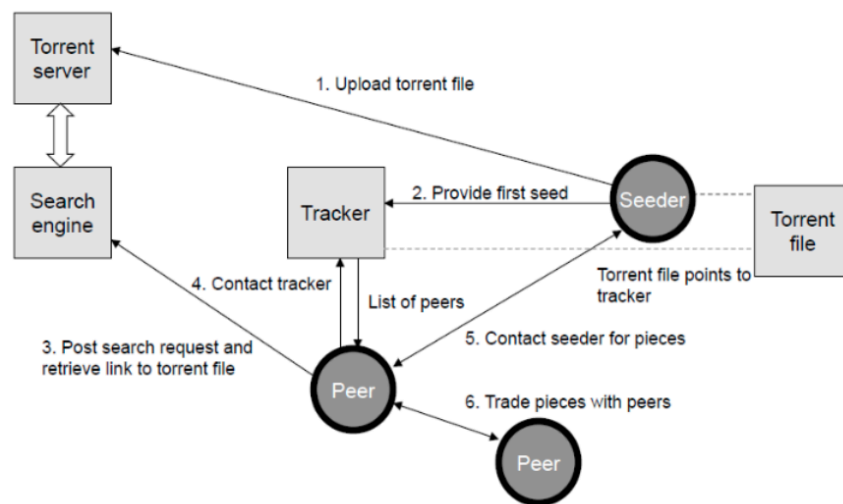


Figure 4.2: BitTorrent protocol overview  
BitTorrent protocol is built on top of HTTP

## Technical Details

- BitTorrent uses a format called **Bencode** to organize its data (not popular like JSON). Bencode supports only 4 data types:

**Strings, Integers, Lists, and Dictionaries.**

- The content is divided into **chunks called pieces** (each one is 256 KB to 2 MB in size).

Once a peer downloads a piece, it becomes a **seeder for that piece** and can share it with others.

- Each **piece** is further split into smaller parts called **subpieces (or blocks)** of **16 KB**.



These blocks can be downloaded **in parallel from different peers**, which helps **speed up the download**.

When all blocks of a piece are downloaded, the **SHA-1 hash** is used to **check for errors** and verify the piece is correct.

- The **Tracker** keeps track of which peers are in which **swarm**, and what part of the file each peer has.

In newer versions of BitTorrent, a **Tracker is no longer needed**—instead, it uses the **Kademlia DHT** to manage connections and find peers. This removes the **centralized point** (the tracker).

## Piece Selection



Figure 4.3: BitTorrent policies over time

The order in which peers choose pieces to download is **very important** for good performance.

If all peers download the same pieces at the same time, they might get **stuck**, unable to exchange useful data with each other.

- **Strict Priority:** A peer **finishes downloading one whole piece** before asking for another. This helps avoid downloading incomplete pieces.
- **Rarest First:** Peers try to **download the pieces that are least common (rarest)** first. Each peer sees how many neighbors have each piece and chooses the rarest.

Why? Because if a peer has a rare piece, **many others will want it**, and this improves the peer's chances of getting faster downloads in return (this is part

of the **"tit-for-tat" strategy**).

- **Random First Piece:** At the **beginning (bootstrap phase)**, a peer picks a **random piece** to download. This is done to **quickly get at least one piece**, so the peer can start **sharing** and **negotiating** with others for the remaining pieces.
- **Endgame:** When the download is **almost complete**, the peer **requests the last few pieces from all peers who have them**, at the same time. This is only done **briefly at the end** to avoid the common problem of **downloads slowing down** just before finishing.

## Free Riders

**Free riders** in BitTorrent are peers who **don't share their upload bandwidth** with others. Some unofficial BitTorrent clients allow users to **limit how much they upload**, which creates a problem because BitTorrent performance **depends on peers helping each other**. When too many people act selfishly, the network slows down.

To solve this problem, BitTorrent uses a **Reciprocity "Tit for Tat"** strategy.

This means: 🖐️ *"You upload to me — I upload to you."* This is done using a technique called **choking**:

- **Choking** = temporarily **not uploading** to a peer.
- Peers decide who to choke/un-choke based on how much the neighbor has uploaded to them.

So, each peer regularly checks:

- How much it is **uploading to** and **downloading from** each neighbor
- Then **chooses a few peers to choke** (and the others stay unchoked)

## Conditions to Receive Data

A peer can receive data **only if**:

1. It is **interested** in the other peer's pieces
2. The other peer has **unchoked** it

## Optimistic Unchoking

If you **only unchoke peers who upload the most**, you ignore **new peers** who might actually be better. To prevent this, BitTorrent uses **Optimistic Unchoking**:

- Every 30 seconds, it **unchokes one random peer** (even if that peer hasn't uploaded yet)
- If this new connection performs well, it might **replace** an existing unchoked peer

## Anti-snubbing Policy

If a peer is **choked by everyone**, it increases the number of **optimistic unchokes** it does (more than just one). This helps it discover new useful connections.

## What About Seeders?

Seeders (peers who already have the full file) **don't need to download**, so "Tit for Tat" does not work for them. Instead, they **unchoke the peers who are uploading the most**, helping the file spread **faster** in the network.

## DHT and BitTorrent

Even though a **tracker** needs very few resources to work, it can still be a **single point of failure**.

It can also be a **bottleneck** (a weak spot) or even a target for **attacks**.

To solve this problem, **BitTorrent Inc.** created its own **Distributed Hash Table (DHT)** called **Mainline DHT**, based on **Kademlia**, but with some improvements in:

- **How the routing table is managed**
- **How the lookups are done**

## Purpose of Mainline DHT

The goal of Mainline DHT is to let peers **find each other** (peer discovery) **without using a central tracker**.

In this case, the DHT works like a **distributed tracker**, where:

- The **key** is the *infohash* of the content (a unique file ID)

- The **value** is the *list of peers* (seeders/leechers) in the swarm

Every node (peer) runs both the **DHT protocol** and the **BitTorrent protocol**.

### Protocol Messages

The DHT protocol uses **four types of messages**:

1. **PING** – to check if a node is online and responding
2. **FIND\_PEER (target ID)** – to search for the node closest to a given ID
3. **ANNOUNCE\_PEER** – to let others know that this peer is now part of a swarm
4. **GET\_PEERS (infohash)** – to find which peers have the file with the given infohash

### How it Works (Example)

The DHT is used just like any normal DHT.

The value you're looking for is the **list of seeders** for a specific **infohash**.

For example:

- **Alice** wants to download a file. She asks her **neighbors** in the DHT if they know who has that file (seeders).
- If her neighbors don't know, they ask **their** neighbors, and so on.
- Finally, **node 15** tells Alice that **node 4** has the file.
- Now Alice connects directly to **node 4** and starts downloading.

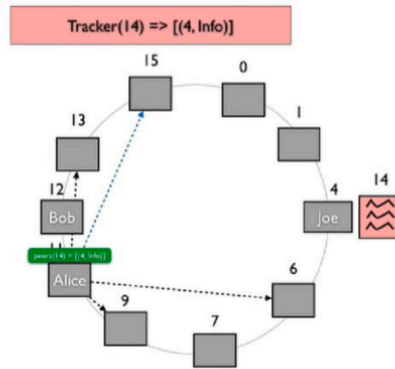


Figure 4.4: 15 has tracking information for the pink (14) content, so it returns info about all the peers providing the file: in this case, only Joe with ID 4

## ✓ Chapter 5: Blockchain

### Basic Concepts of Blockchain

- Ledger
- Consensus in a distributed environment
- Tamper freeness
- Proof of ownership
- Permissioned and permissionless blockchains

Each **block** in a blockchain contains:

- Some **data**
- A **hash** (like a digital fingerprint)
- The **hash of the previous block**

The **link** to the previous block makes sure that all blocks are in the correct **order**, forming a **chain of blocks**.

**Tamper freeness** means that if someone tries to **change one block**, the **hash** of that block changes. Because the next block includes the old hash, its hash must

also change — and so on for all blocks that follow. This makes changing any block very hard, because you would also need to:

- Recalculate many hashes
- Find a new valid **Proof of Work** (a difficult math problem)

A **ledger** works like a **notary** — it keeps track of events and their order.

In blockchain, this ledger is **shared and copied** on every peer (node) in the P2P network. It is **unchangeable** and protected by the tamper-proof property.

Think of the ledger like a **bulletin board** that stores every operation in the order they happened. It only allows adding new records (append-only), and no one can delete or change old records. When the ledger is **organized in blocks**, we call it a **blockchain**.

## Consensus and Challenges

**Consensus** is the method used to decide **which operation** will be added to the blockchain and **who** gets to make that decision.

It works like a **voting system**, but there are some problems to deal with, like preventing:

- **Double spending** (using the same money twice)
- **Fake votes**

A common problem in voting is the **Sybil attack**, where one computer pretends to be many fake users to get more votes.

One solution is to use **Proof of Work (PoW)** to make cheating very hard.

In PoW, nodes have to **solve a very hard math problem** to vote.

This makes it **too expensive or difficult** for one person to fake many votes — preventing Sybil attacks.

**Proof of Work** is like a **lottery**, where:

- Each ticket costs **computational power** (i.e. effort from your computer)
- The **first one to solve** the problem wins and gets to **add the next block** to the blockchain

If **two nodes win at the same time**, we get a **fork** (two versions of the chain). The system then picks the fork that becomes **longer**, because it's backed by **more total work (power)**.

### Restricted Access (Permissioned Blockchain)

A **permissioned blockchain** is one where **only selected nodes** can vote and add new blocks.

This is useful when, for example, a **group of companies** wants to share a blockchain for their own private use.

This kind of blockchain can be used to:

- **Prove the supply chain** of a product is correct (where it came from)
- **Track the full history** of a product — from raw materials to the final item
- **Find out** at what point in the supply chain something went wrong or was changed

## Chapter 6: Tools for DHT and Blockchains

### Cryptographic Tools

**Definition 6.1 (Hash Function):** A **hash function** takes a binary input (a string of 0s and 1s) of any length and turns it into a **fixed-length** binary output.

### Hash Functions and Collisions

Regular (non-cryptographic) hash functions are designed to have a **low chance** of collisions (two inputs producing the same output). But for an attacker who **tries on purpose** to create a collision, some functions are **too weak**.

For example, **CRC (Cyclic Redundancy Check)** was once used to check data correctness, but it's not secure for **cryptographic purposes**. Even though random errors rarely cause a collision, an attacker can **easily create one** with some effort.

In general, **collisions are always possible** in hash functions. This is because the number of possible inputs is **bigger** than the number of possible outputs.

**Hash security** means: How hard is it for someone to **find two inputs** that create the **same hash**?

## Cryptographic Hash Functions

For a hash function to be **secure in cryptography**, it must have two main properties:

1. **Collision resistance** — it must be **very hard** to find **two inputs** that give the **same hash**
2. **One-way property** — given the output of a hash, it must be **very hard to find the input**

## Hiding and Puzzles

In **blockchains and cryptocurrencies**, we also need two more properties:

**Definition 6.2 (Hiding):** A hash function  $H$  is **hiding** if: If someone chooses a **secret value  $R$**  from a **strong random source**, then even if you are given  $H(R||x)$ , it's still **almost impossible to figure out  $x$** .

This means attackers can't learn secret information just from the hash, even if they know the output.

Bitcoin Proof of Work (PoW) is based on a hash/search puzzle.

## (Puzzle friendliness)

A function  $H$  is called **puzzle friendly** if:

- For every possible output value  $y$  that is  $n$  bits long, and if  $r$  is chosen randomly from a distribution that is very hard to guess (high min-entropy), then it is **very hard** to find an  $x$  such that  $H(r||x) = y$ , without trying almost all possible  $x$  values — meaning, it takes about  $2^n$  tries.

The **puzzle-friendly** property means that there is **no shortcut** to solve such search puzzles — the only good way is to **try all possible values of  $x$  one by one (brute force)**.



## Bloom Filters

A **Bloom Filter** is a tool used to answer questions like: “Is element **k** in the set **s**?” It is used to solve the **Set Membership** problem. Bloom Filters are **very fast and lightweight**, but they give **probabilistic** answers — meaning they can give **false positives**, but **never false negatives**. A common use of Bloom Filters is to **check intersections** between them. You can also compute the **union** of two Bloom Filters by using the **bitwise OR** between the two filters.

Bloom Filters **cannot delete elements**, but a **Counting Bloom Filter** can be used instead. It tracks how many times an element was added by **counting**, so it can **increase or decrease** the count as needed.

**Bloom Filters are used in real systems** like **Ethereum**, **Google**, and **Bitcoin**.

## Merkle Hash Tree

A **Merkle Hash Tree** is a data structure that helps to **summarize a large amount of data**, and **verify** if the content is correct. It is a **binary tree** of hashes built like this:

- The **i-th leaf** contains the hash **hi** of a piece of data **fi**
- Every **internal node** contains the **hash** of the **concatenation** of its two child hashes
- The final hash at the **top (root)** is called the **Merkle Root Hash**

This structure is often used to **verify** that a piece of data belongs to a set, **without storing the whole set**.

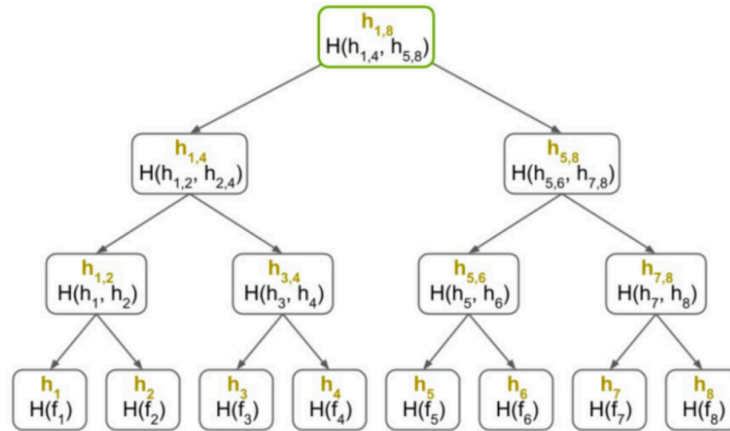


Figure 6.1: Merkle

### Tries and Patricia Tries: Tries:

- The **root node** stores nothing
- **Edges** (connections) are labeled with **letters**, and a path from root to a node forms a **string**
- Each **node** has a marker to show if it is the **end of a valid string**

However, **tries use a lot of space**, since each node stores a label.

To **save space**, we can compress tries by storing only the **first different parts** (prefixes) in each node — this results in a **Patricia Trie** (also called a compact trie).

### Patricia Merkle Trie

A **Patricia Merkle Trie** is a combination of:

- A **Merkle Tree** (for verifying data)
- A **Patricia Trie** (for compressing strings)

This structure is **used in Ethereum** to store important blockchain data.

### There are three types of nodes:

#### 1. Branch Node

- Has **many children**
- Can store a value if the key ends at this node (otherwise, it's just a prefix)

## 2. Extension Node

- Has **only one child**
- Stores a **part of the key (a prefix)**

## 3. Leaf Node

- Stores the **last part of the key** and the **final value**

A trie stores **key-value pairs**:

- The **key** is a string
- The **value** is either a **pointer** to another node or the **actual data** (in leaf nodes)

In **Ethereum**, the Patricia Merkle Trie is used to store:

1. **Accounts** → where the key is the account address, and the value is the **account balance**
2. **Transactions** → where the key is a transaction and the value is the **amount transferred**

Since nodes can contain **multiple elements**, and hash functions work on strings, the data must be **serialized** (converted to a string format) before hashing:

- **HP (Hex Prefix Encoding)** is used for keys
- **RLP (Recursive Length Prefix Encoding)** is used for values (which may be more complex data)

# Chapter 7: Bitcoin Transactions and Scripts

In the **late 1990s**, there was an early idea for digital money called **e-Cash**.

**e-Cash** introduced a concept called **blind signatures**, which allowed a **bank** to sign a transaction **without knowing its details**.

The idea went **beyond normal public key cryptography**. A **nonce** (a random number used once) was added and **mathematically combined** with the transaction data to make it unreadable — this became the “**scrambled data**.”

The **bank would then sign** this scrambled data (a “blind signature”), but could **not see the original information** inside it.

## Bitcoin Release

In **2008**, **Satoshi Nakamoto** introduced **Bitcoin**, a **decentralized digital currency** with the following features:

- **Double-spending** is prevented using a **peer-to-peer (P2P) network**
- There is **no mint** or central trusted party
- Users can remain **anonymous**
- New coins are created using **Proof of Work**, similar to **Hashcash**
- The same **Proof of Work** used to create new coins also helps prevent **double-spending**

**Bitcoin** is **different from e-Cash** because it is **fully decentralized**, with **no central authority** like banks. It runs **entirely on a P2P network**.

## Bitcoin Identity

In **cryptographic systems** like Bitcoin, creating a new identity is easy:

Just generate a **new random key pair**, which includes:

- A **private key (sk)** – your secret
- A **public key (pk)** – your public name

To **verify** a signed transaction, the system checks: **verify(pk, data, sig) == true**

This means that the signature **sig** was created using the matching private key of **pk**.

In most cases, a **Bitcoin address** is made from a **public key** and represents the **owner** of the key pair. But sometimes, it can also be a **script**.

Anyone can create a **new identity** (a new key pair) **at any time**.

These identities are **not linked to real-world names**, but:

- Since the **blockchain is public and readable by anyone**,
- People can **watch the activity** of an identity over time,
- And may be able to **guess who it belongs to**.

## Bitcoin Transactions

A **transaction** (t) is used to **send money** from the **sender** to the **receiver**, and it is **added to the blockchain** once it is **confirmed**. The transaction (t) is first **sent to the network** (broadcast). Then, **miners** collect these broadcasted transactions and add them to a **candidate block**. When a **miner solves the Proof of Work**, the **block is shared** with the network. If this block **includes t**, then the transaction t is considered **confirmed**.

Each transaction has:

- On the **left side**: a list of **inputs** (where the money comes from)
- On the **right side**: a list of **outputs** (where the money is sent)

For example (like in Fig. 7.3):

- **Bob receives 0.25 BTC → Alice provides 1.00 BTC as input → Alice gets 0.75 BTC back as change**

Every transaction must be **balanced**:

- The **total input** must **equal the total output**,
- But the difference between input and output is the **transaction fee**
- This **fee** goes to the **miner** who confirms the transaction.

Transactions can also:

- **Combine money** from **many inputs into one output** (merge)
- **Split money** from **one input into several outputs** (distribute)

So, Bitcoin transactions can have **multiple inputs and multiple outputs**.

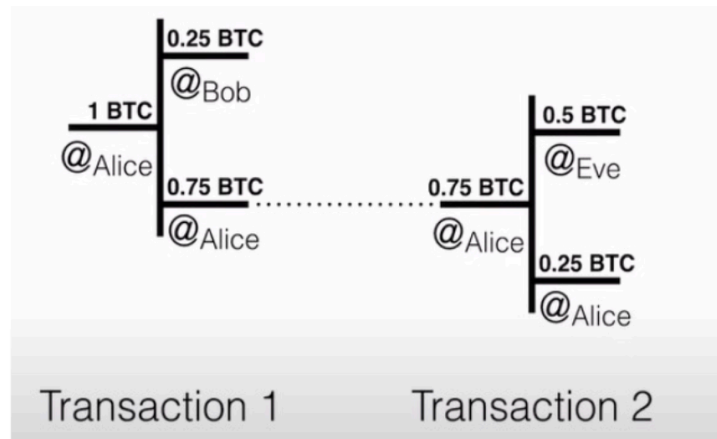


Figure 7.3: Bitcoin linked transactions

Each transaction has a **script**, which is written in a simple and purposely limited programming language.

This script's job is to **check who owns the funds being transferred**.

Running the script:

- Has **no memory** (stateless)
- Always gives the same result for the same input (deterministic)

The most common use of the script is to **check a digital signature**.

There are different types of scripts:

- **Pay to Public Key (P2PK)**: the simplest case
- **Pay to Public Key Hash (P2PKH)**: the most common case
- **Pay to Script Hash (P2SH)**
- **Pay to Multi-signature**

### Simplified locking and unlocking

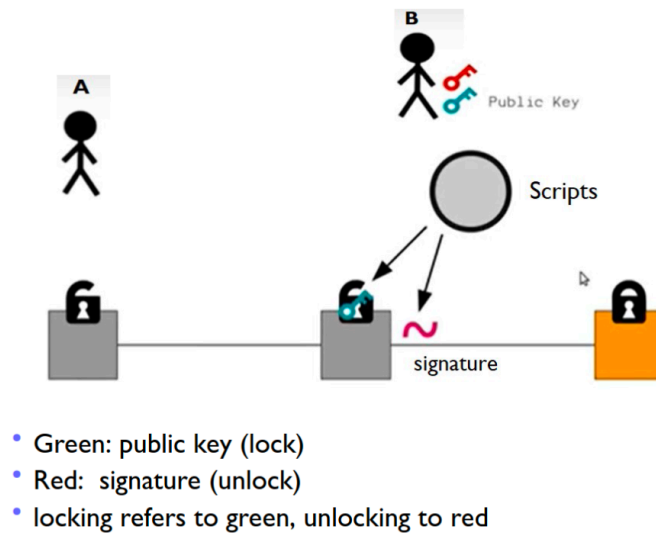


Figure 7.8: Bitcoin script example

#### Put simply

B first notifies its public key to A, who now knows to whom to send the bitcoin. A unlocks some of its bitcoins—locked by a previous transaction—and creates a lock on the bitcoin sent to B, which can be unlocked by B using its private key.

## Analyzing P2PKH

As mentioned earlier, a Bitcoin transaction works like this:

- The **sender adds a script** to the bitcoin being sent. This script **locks** the bitcoin so that only the **receiver** can use (spend) it.
- This script is called the **locking script** and is saved in the UTXO (Unspent Transaction Output).
- The **receiver**, when they want to spend the bitcoin later, will write an **unlocking script** (usually a signature).
- The unlocking script is added **in front of** the locking script taken from the UTXO.
- Then, the **Bitcoin network runs the combined script** to check if the person trying to spend the bitcoin is really allowed to.

The **locking script** is saved in the transaction's output and will be used by the receiver to spend those funds later.

Running both scripts together (unlocking + locking) allows the receiver to actually use the bitcoin.

In the case of **P2PKH (Pay to Public Key Hash)**:

- The **locking script** locks the bitcoin to the **receiver's address** (which is the hash of their public key).
- The **unlocking script**, in most cases, is a **signature** of the whole transaction (inputs and outputs).
- But this isn't always required — it can vary based on the script type.

### Coinbase Transaction

A **coinbase transaction** is a special transaction that creates **new bitcoins**. It's used to **reward miners** for solving the Proof of Work.

It is not linked to any previous transaction (no inputs), because the bitcoins are newly created by the system.

### Transaction Lifecycle

1. The transaction is **created**
2. It is **signed** using one or more **digital signatures** to approve spending the funds (see section 7.3.2.2 for signature types)
3. The signed transaction is **broadcasted** to the Bitcoin **P2P network**
4. Every **node** (participant in the network) **checks the transaction** and shares it with others, so it spreads across the network
5. A **miner** checks and **adds the transaction to a block**
6. The block is added to the **blockchain**. Once it has enough **confirmations** (more blocks added after it), the transaction is **permanently recorded**
7. The funds are now considered **owned by the new user**, and that user can now **spend them** in another transaction — continuing the chain of ownership

## Chapter 8: Bitcoin Mining



Distributed consensus is a method used to make sure all nodes (computers) in a **decentralized system** agree on the same data, like the state of a blockchain.

This agreement must work **correctly even when** some nodes are broken, the network is unstable, or some nodes try to cheat (called **Byzantine faults**).

## Competing

Every node (computer) has a **MemPool** – a list of Bitcoin transactions that are waiting to be confirmed.

If two transactions try to spend the same bitcoin (called **double-spending**), the node will **keep the first one** it received and **ignore the second**. Nodes compete to **add transactions** to the blockchain.

Bitcoin uses a system called **Nakamoto consensus**, which works like a **lottery**. The winner of the lottery **adds a block of valid transactions** to the blockchain and then **shares** the new version of the blockchain with other nodes. This competition is called **mining**.

## Mining

Mining starts when a miner **chooses some transactions** from the MemPool and puts them in a **candidate block**. Then, the miner creates the **block header** (which is about 1000 times smaller than the full block).

Finally, the miner performs the **Proof of Work**, a hard puzzle that requires a lot of computing power. So, while **anyone** can create a transaction, **only miners** can build blocks. Each block includes **many transactions** and a **block header**.

The block header contains:

- Version
- Timestamp
- mhash The Merkle root of the transactions in the block Merkle tree is not explicitly represented in the block, it is built on demand.  
Any change to a transaction results in a change of mhash, consequently in a change of the hash of the whole block.
- hashprev The hash of the previous block header, and a hash pointer to the previous block
- PoW related fields: Target & Nonce

## Consensus

A node is chosen to propose the next block based on a resource that is hard to control by one party.

In **Bitcoin**, this resource is **computational power**, and the choice is made using **Proof of Work (PoW)**.

The **consensus is implicit**, meaning:

- There is **no voting** or shared algorithm run by all nodes.
- The system **automatically handles** the selection of nodes, even if some are malicious.
- Even if nodes sometimes see **different versions of the blockchain** (called forks), they will **eventually agree** on one version.
- The correct version will be the **longest chain**, as it represents the most total work.

## Proof of Work

- **d** – difficulty: a number that adjusts how hard it is to solve the PoW.
- **c** – challenge: a string (usually the block header without the nonce).
- **x** – nonce: a number the miner tries to guess.

A **Proof of Work function** is written as:

**Fd(c, x) → True or False**, and must satisfy:

1. **d** and **c** are given.
2. If **d**, **c**, and **x** are known, it is **fast to check** if the result is correct.
3. But, **finding x** that gives **True** is **computationally hard**, though possible.

The PoW is hard because the result looks like a **random 256-bit number**, where each bit is like a **coin flip** (0 or 1).

So, there is **no shortcut** — miners must **try many guesses (brute force)** to find the right nonce.

The **probability** **p** that a guessed hash is **below the target T** is:

$$p = T + 1 / 2^{256}$$

The **average number of tries**  needed to find a valid hash is:

$$a = 1 / p = 2^{256} / (T + 1)$$

## Resistance to Sybil Attacks

The system is protected from **Sybil attacks** (where one attacker creates many fake identities) because:

- What matters is not the **number of identities**, but the **total computing power**.
- Faking votes would require huge computational resources, which is **very expensive**.

## Block Propagation

After a block is mined, it is **broadcast to the network**.

Each node that receives the block will:

- **Check the Proof of Work** by hashing the block header.
- Make sure the **hash is smaller than the target**.

This check is **easy to do** and doesn't require any central authority.

Once verified, the node:

- **Adds the block** to the blockchain.
- **Removes any conflicting transactions** from its memory pool (MemPool).

## Incentives

There are **two ways** to **reward miners** for being honest:

### 1. Block Reward

- A **payment to the miner** for creating a new block.
- When a block is mined, **new bitcoins are created** — this is the **only way** new bitcoins come into existence.
- The reward is **cut in half** every 210,000 blocks (about **every 4 years**).
- The **last reward** will be given in the year **2140**.

## 2. Transaction Fees

- For each transaction in a block, the miner keeps the **difference between inputs and outputs** as a fee.
- These fees were added to **motivate miners** to include transactions and give better service.
- The **first transaction** in every block is called the **coinbase transaction**.
  - It **creates new bitcoins** and includes the **block reward plus all transaction fees**.
  - It is not linked to any previous transaction (UTXO), but to a **"dummy" input**.

## Mining Difficulty

"To deal with faster computers and changing number of miners, the difficulty of Proof of Work changes regularly to keep the block rate stable."

– Satoshi Nakamoto

- The **difficulty** is adjusted every **2016 blocks** (about every **2 weeks**).
- The goal is to keep the **average time to mine a block** close to **10 minutes**.
- The difficulty is changed by updating the **target**:
  - **Higher difficulty → lower target → harder to find a hash.**
  - **Lower difficulty → higher target → easier to find a hash.**

## Why 10 Minutes?

"If block broadcasts are slower than expected, we may need to make blocks less frequent. We want blocks to spread through the network quickly, before the next block is mined."

– Satoshi Nakamoto

- The **10-minute block time** is a **trade-off**:
  - Blocks need **enough time to reach all nodes** in the network.

- If blocks are mined too fast, some nodes might work on **old (obsolete) blocks**, wasting time and energy.

## Tamper-Freeness

The blockchain cannot be easily changed (tamper-free) because:

- The **Proof of Work (PoW)** is **very hard to solve**
- But it is **easy to check** if a PoW is correct
- Doing PoW needs **a lot of resources** (like time and energy), which are **limited**

## Temporary Forks

Temporary forks can happen when **two miners** solve a block at the same time and **send it to the network**.

- This creates **two branches** of the blockchain from the same parent block.
- This is **not the same** as double-spending, but it does raise the question:

**Which version of the blockchain is correct?**

Each miner might see **block A or block C first**, and will **start mining on the one they received first**.

As a result, the two forks may grow separately for a while.

- If a miner later receives a block that makes the **other chain longer**, it **abandons the shorter one**.
- Transactions in the **abandoned chain** (that are not in the longer one) are returned to the **MemPool** to be used again.

→ **Confirmation Rule**: A transaction is **fully approved** only when **5 or more blocks** come after it.

→ **Nakamoto Consensus**: Forks are eventually resolved, and **all nodes agree** on the **longest chain**. This gives the system **eventual consistency**.

## Mining Technologies

The main actors in the Bitcoin network are:

1. **Reference client:** Bitcoin Core
2. **Full blockchain mode:** stores and verifies the full blockchain
3. **Solo miner:** mines independently
4. **Lightweight (SPV) wallet:** doesn't store full blockchain, relies on full nodes

### Centralized Mining Pools

Mining is **risky**, because you may spend a lot on **hardware and electricity** and still not get a reward for a long time.

**Mining pools** are a way to **share both the risk and the reward** among many miners.

- A **Pool Manager** sends work to the miners and **divides the reward** based on how much work each miner does.
- The **manager must be trusted** by everyone in the pool.

### **Problems:**

- How can the manager **measure the work** done by each miner?
- Even miners who didn't find a block should be rewarded.
- Some miners might **cheat**, claiming they worked more than they really did.

To fix this, miners send "**near-valid blocks**" (shares) to show they are working.

### Reward Systems:

- **Pay-Per-Share (PPS):**
  - Miners get paid **a fixed amount** for each share they submit.
  - The **pool pays** from its own balance.
  - The miner who finds the block does **not get extra** reward.
- **Pay-Proportional:**
  - Miners get rewarded **based on the amount of work** they've done.
  - This **encourages** submitting valid work.

- The pool **only pays** when a real block is mined, so it's **less risky** for the manager.

### Decentralized Mining Pools

The idea is to **remove the central manager** and **share rewards fairly** using a new system:

- Miners create a **private chain** of “**weak blocks**” (with lower difficulty).
- These weak blocks include transactions that reward the miners.
- When a miner finally finds a **valid block**, the **reward is given** using a side blockchain.
- This side chain is then **merged with the main blockchain**.

#### **Advantages:**

- No need for a **central pool manager**
- Rewards are given **fairly and transparently**
- Keeps the system **efficient** with **little extra overhead**

## Chapter 9: Bitcoin Attacks

Simple attempts to double spend are **easily noticed and blocked** by the network. If someone tries to send **two transactions using the same coin**, the network will:

- Accept **only the first one** that it receives.
- **Reject the second one** because the coin was already spent.

Also, if someone tries to **add the same coin twice** in a single block, the network will reject the **whole block** as invalid. However, there are also **more advanced double spending attacks**

### **Forks**

A **fork** happens when **two blocks (A and B)** are mined at almost the same time.

- This happens because **block B is mined before block A is fully shared** across the network.
- Some nodes will see block **A as the latest block**, and others will see **B**.
- So, we get **two versions** of the blockchain at the same time.

The fork continues **until the next block is mined**:

- If the next block is mined on top of **A**, then A's chain becomes the **longest** and is accepted as the **main chain**.
- The version with block **B** is **abandoned**, and the network updates (reorganizes) the chain.

The same would happen the other way around (if the next block is mined on top of B).

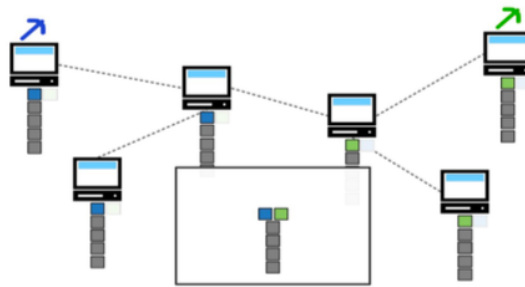


Figure 9.1: Bitcoin Fork

### Forks and Double Spending

If someone tries a **double spending attack**, they send **two transactions using the same coin**.

- Usually, **only one** of these gets mined — the other is **invalid** and will be ignored by the network.

But if both transactions are included in **two different blocks** mined at the same time, it creates a **fork**.

In that case, **both transactions could exist temporarily**, one in each fork.

To prevent this from causing problems, the network **accepts a transaction only after 6 new blocks** are added **on top of it**.



➔ This rule helps make sure the transaction is **secure** and won't be removed later due to a fork.

## 51% Attack

This attack is a way to **double spend** by gaining control of **more than 50% of the total mining power** in the Bitcoin network.

- The attacker builds a **"secret" fork** of the blockchain that **does not include** the transaction where he sent coins.
- Meanwhile, on the **public blockchain**, he sends coins to someone (for example, to buy something).
- Once his **secret chain becomes longer** than the public one, he **broadcasts it** to the network.
- Because it's the longest chain, the network accepts it — and the coins are **not shown as spent anymore**.

➡ This means the attacker **gets back the coins** while still having received something in return (like goods or services).

But this attack is **very hard to do** in real life — it's **very unlikely** because controlling **more than 50% of mining power** is extremely expensive.

## Transaction Malleability

This is a **more subtle attack**, and it's **not a double spend** — but it can be used to **confuse users or the system**.

- Remember: in Bitcoin, each transaction is **signed** using the **private key** of the owner, and other nodes check that the person is allowed to spend those coins.
- A smart attacker can **change the unlocking script** (like reordering things or adding a "do nothing" operation), so that:
  - The **transaction still works**,
  - The **signature is still valid**,
  - But the **transaction ID (TXID)** changes!

This works because TXID is made by **hashing more data** than what the signature covers. So the **same transaction** can get a **different TXID**, without changing what it does.

### Exploiting Malleability

Let's say **Alice sends Bitcoin to Bob** in a transaction called **TXID1**.

- **Before TXID1 is confirmed**, Bob **modifies** the signature and creates a **new version** with a different TXID, called **TXID2**.
- If **TXID2 gets confirmed first**, TXID1 will be **rejected**, because the coins have already been spent.

Now Alice sees that her **original TXID1 never confirmed**, so she thinks the transaction **failed** and may **send the payment again**.

➡ In the end, Bob **gets paid twice**.

### Segregated Witness (SegWit)

SegWit was added to Bitcoin in **August 2017** to **fix transaction malleability** and **increase the block size**.

- It **separates the signature** from the rest of the transaction.
- The **signature data** goes into a **new structure** called the **witness**.
- Since this part is **not used to compute TXID**, changing the signature does **not change the TXID** anymore.

This **stops the malleability problem** and allows Bitcoin to work more efficiently.

### Lottery is Costly

Trying to **cheat** in the Bitcoin system is **very expensive** — it wastes energy and money. But even **honest mining** uses **a lot of energy**, and the **reward is not guaranteed**. This is why most miners **join mining pools** — groups of miners that **share rewards** based on how much computing power they contribute.

### Mining Pool Reward Methods

There are different ways to **reward miners fairly** inside a pool:

### Pay per Share (PPS)

- Miners are **paid instantly** based on the number of shares they contribute.
- The **pool operator** takes the risk — they might **not find a block**, but still have to **pay the miners**.

### Pay Proportional

- Miners are **paid only when a block is found**, based on their contribution.
- The **operator takes no risk**, but miners must **trust the operator** to be fair.

### Decentralized Mining Pool

- Uses a **separate private blockchain** (called **sharechain**) to track miner contributions.
- In the sharechain:
  - A new block is created every 30 seconds (with **low difficulty**).
  - Miners mine **"weak blocks"** (partial solutions of PoW).
  - All miner activity is **public and transparent**, so **no one can cheat**.

When a miner finally solves the **real Proof of Work**, that block is added to the **main Bitcoin blockchain**, and the reward is **shared fairly** based on sharechain records.

## Chapter 10: Bitcoin Scalability

### Blockchain Trilemma

The **Blockchain trilemma** says that a blockchain can only achieve **two out of these three things**:

- **Decentralization**
- **Security**
- **Scalability**

These three goals often **conflict** with each other. The idea was first introduced by **Vitalik Buterin**, the founder of Ethereum.

### On-chain Scalability

**On-chain scalability** means the blockchain's ability to process **many transactions per second (TPS)**.

- **Bitcoin** can handle **7 TPS**, and **Ethereum** can handle **15 TPS**.
- These numbers are **very low** compared to systems like **Visa**, which can handle **24,000 TPS**.

Also note:

- Even though Bitcoin handles 7 TPS, the transactions are **not confirmed immediately**.
- A Bitcoin transaction is considered **confirmed only after 6 blocks**, which takes around **1 hour**.
- The **average transaction fee** for a 6-block confirmation is about **\$55**.

**On-chain scalability solutions** (each with pros and cons):

- ♦ Increase **block size**
- ♦ Increase **block frequency**
- ♦ Use **cross-chains** and **side-chains**
- ♦ Use **alternative consensus methods**, like **Proof of Stake**

### Off-chain Scalability

**Off-chain scalability** means moving some transactions **outside the main blockchain**, and **only using the main blockchain when needed**.

- This reduces the number of transactions on the main chain.
- The main chain acts like a **referee**, used only to settle disputes or finalize balances.

Think of off-chain transactions like **IOUs** (promises to pay). These IOUs are settled on the main blockchain **later**.

- These transactions are **almost as secure** as on-chain ones.
- They allow **fast, cheap**, and **micropayments**, with **no need to trust** the other party.

#### **Possible issues with off-chain solutions:**

- ♦ Risk of **centralization**
- ♦ **Locked funds** (you can't use them elsewhere)
- ♦ The system must be **always online**

✅ **Bitcoin's Lightning Network** is an example of off-chain scalability:

- It's a **second layer** built on top of Bitcoin.
- Allows **instant, low-fee**, and **micropayments**.
- It can currently handle **10,000 TPS**, and it's growing.

### **MultiSignature Transactions**

**MultiSignature (MultiSig)** transactions require **multiple signatures** to spend the coins. This makes the transaction **more secure**.

- Useful especially for **off-chain transactions**, where both parties or a third party must agree before the money moves.

These are created using the **CHECKMULTISIG** script, which checks that enough valid signatures are present.

#### **Examples of MultiSig setups:**

- **1-of-n**: Only **one signature** is needed out of multiple people.
- **2-of-2**: **Both parties** must sign to spend the funds.
- **2-of-3**: Any **two out of three** must sign.

### **Escrow Contracts**

An **escrow contract** uses a **trusted third party** to hold the money **until certain conditions are met**.

Let's say **Alice wants to buy** something from **Bob**, but she doesn't trust him.

They can use a third party, **Judy**, to act as the **escrow**:

1. Alice creates a **2-of-3 MultiSig** transaction with **herself, Bob, and Judy**.
2. The funds are **locked** until **two out of the three** people agree where to send them.
  - If there's **no problem**, Alice and Bob can **release the funds together**, without Judy.
  - If there is a **dispute**, Judy helps **decide who gets the money**.

 This system requires **two transactions**:

- One from Alice to the **escrow**.
- One to send the coins to **Bob or back to Alice**.

### Towards Pay-to-Script-Hash (P2SH)

**MultiSig scripts** are usually **long and complex**, which:

- Requires **many public keys**, Makes transactions **bigger**, Increases **fees**.

**P2SH** simplifies this process:

- The **complex script is hidden** by **hashing it**, and only the **hash** is used as the address.
- The **script is only revealed** when the money is spent.
- The **sender doesn't need to understand** the full script — it's simpler and cleaner.

 Benefits:

- **Shorter address, Lower fees, Less complexity for the sender**

### Hash-Time Locked Contracts (HTLCs)

**HTLCs** are special contracts that lock the funds:

- Until a certain **time passes**, or
- Until a certain **condition** is met.

They are used in the **Lightning Network** to make off-chain payments **secure**.

### Clients vs Full Nodes

- A **Full Node**:
  - Stores the **entire blockchain** (about **500 GB**),
  - **Validates every transaction**,
  - Is the **most secure** way to use Bitcoin,
  - But requires **a lot of space and power**.
- Many users use **Simplified Payment Verification (SPV)**:
  - Only stores **block headers**, which are **1,000 times smaller**.
  - Can **verify transactions** without needing the whole blockchain.
  - Less secure than full nodes.

✅ How SPV works:

- It uses a **Merkle Proof** to check if a transaction belongs to a block.
- The **Merkle Root** from the proof must match the one in the **block header**.

## ✅ Chapter 11: Bitcoin Lightning network

Bitcoin is **not ready to handle** the world's financial transactions.

- It only processes around **250,000 transactions per day**, while **VISA** can handle about **47,000 transactions every second**.
- This is mostly because:
  - The **maximum block size** is only **1MB**

- A new block is created every **10 minutes**
- Each transaction must be **broadcast to all nodes**

**BitcoinCash** was created (a "fork" from Bitcoin) to solve this by:

- Increasing block size from **1MB to 8MB**, and later to **32MB**.

But this is **not a perfect solution**:

- Block size can't grow **forever**
- Bigger blocks mean more data for each node to store
- This would make the network **more centralized**, because each node would need to store around **400TB** of data every year.

### Introduction to the Lightning Network

The **Lightning Network** is a **second layer** built on top of the Bitcoin blockchain.

- In this layer, transactions are **not broadcast** to the network.
- Instead, they are only recorded on the main blockchain **when needed**.

Here's how it works:

- A customer deposits money into a **payment channel**.
- Payments happen on a **private ledger**, managed by the merchant.
- Each time the customer buys something, a **new transaction replaces the previous one**.
- The merchant takes **no risk**, because he can **record the latest transaction on the blockchain** at any time.
- The customer can **only spend** what he already deposited.

Opening a channel is itself a blockchain transaction, so it's **not practical** to open a new one for every payment.

That's why Lightning uses a **network of connected channels**, so you can send payments through **existing routes**.



## How it Works

### 1. Opening a Lightning Channel

Example: Alice wants to pay Bob.

- Alice or Bob (or both) puts Bitcoin into a **2-of-2 MultiSig wallet**.
- Think of it like **opening a bar tab**: you pay upfront, and can do many transactions later.
- Alice sends Bitcoin into this shared wallet to fund the channel.

### 2. Transacting in the Channel

- Alice can now send money to Bob **off-chain** (not recorded on the blockchain).
- This means payments are **fast and cheap**.
- Both Alice and Bob have a **signed copy of the current balance** in the channel.

### 3. Closing the Channel

- When they're done, they **close the channel**.
- This **final balance** is then saved on the blockchain.
- Like **paying your final bill** at a bar when you leave.

💡 If Bob disappears after Alice has funded the wallet, **Alice can get her money back after 30 days**.

## Punishing Cheaters


A problem in Lightning is that **older transactions are still valid**, even if there are newer ones.

So, if Alice tries to cheat by **publishing an old transaction**, we need a way to **punish** her.

🔒 Solution: Use **revocation secrets**:

- Every time Alice makes a new transaction, she sends Bob a **secret code** (the revocation secret) for the old one.

- If Alice tries to cheat and publish an old transaction, **Bob can use the secret to take all the funds.**

 To make this work, there's usually a **24-hour delay** before the transaction is final, giving Bob time to react.

## Routing Payments

The Lightning Network (LN) uses a source routing algorithm. This means the sender (each node, actually) knows the whole network and can find the shortest path to the receiver.

To protect privacy, Alice can use **Onion Routing**. In this method, the payment is encrypted in many layers. Each node along the path can only remove one outer layer of encryption.

Channels have fixed capacity, which is set when the channel is first created. So the network needs to stay balanced to allow payments in both directions.

If the payment fails because a channel doesn't have enough capacity, the money is returned to Alice, and she tries a different path.

## Doubts and Open Problems

It has not been proved yet that the Lightning Network is Scalable, Decentralized and Secure at an acceptable level.

Many problems are still open:

- You cannot just "enter the network", you need a payment channel, which is costful.
- Longer the route, longer the chances of delay
- Need for routing algorithms
- Counterparty miss behaviour can leave coins locked for a long time

## Chapter 12: Ethereum

**Ethereum** is a **decentralized platform** that runs **smart contracts**.

Smart contracts are computer programs that run exactly as written — no downtime, no fraud, and no interference from third parties.

Ethereum is not just for cryptocurrencies. It is a blockchain platform for building **decentralized applications (dApps)** such as:

- Crowdfunding, Tokens, Self-sovereign identity (SSI), Supply chains & etc

### What is a Smart Contract?

A smart contract is a **computer program** that automatically executes the terms of a contract.

Its main goals are to:

- Make sure contract conditions (like payments, confidentiality, enforcement) are followed
- Reduce mistakes or cheating
- Reduce the need for trusted middlemen
- Lower fraud, arbitration, and enforcement costs

Smart contracts are written in a programming language called **Solidity**.

They work like "if this happens, then do that" instructions in a contract.

Ethereum's blockchain supports **distributed data storage** and **computations** for smart contracts.

### Ethereum vs Bitcoin Scripting

Ethereum's programming language is different from Bitcoin's in several ways:

- **Turing completeness:** Ethereum can solve any computational problem (Bitcoin's scripting is limited).
- **State:** Ethereum programs can remember past information (Bitcoin scripts cannot).

- **Blockchain awareness:** Ethereum contracts can read blockchain data like block headers.

## Features of Smart Contracts

- Everyone on the blockchain runs the **same code** and verifies it.
- Smart contracts must be **deterministic** (always produce the same output from the same input).
- Contract logic is **visible to all participants** (transparency).
- Privacy can be a concern because of this visibility.
- Since they are Turing complete, smart contracts can do anything a computer can do.
- Running smart contracts costs money because all nodes must run the code in parallel.
- Nodes are **paid (rewarded)** for executing smart contracts.

## Ethereum State Machine

Bitcoin stores its state in **UTXOs** (unspent transaction outputs).

Ethereum stores its state in **accounts** which keep track of balances.

Ethereum has a **transaction-based deterministic state machine** that changes state through transactions. Users can create their own state transition functions that trigger state changes.

## Ether and Accounts

- **Ether (ETH)** is Ethereum's official cryptocurrency. It pays for transaction fees and computational work.
- **EOAs (Externally Owned Accounts):** Controlled by private keys and can send transactions to other accounts. These act as a bridge between the outside world and Ethereum's internal state.

- **EVM (Ethereum Virtual Machine):** The runtime environment where smart contracts run. It is isolated from the network and computer systems.
- To prevent infinite loops or attacks, Ethereum uses a **gas mechanism**.

## Gas and Fees

- **Gas** is the fee you pay for executing smart contracts.
- Gas price varies: low price = slow processing, high price = fast processing.
- You set a **gas limit**, the maximum gas you want to spend on a transaction.
- If your gas runs out during execution, the transaction stops, state changes revert, but you still pay the gas fee to miners.
- Miners earn gas fees and prefer transactions with higher gas prices.

## The Merge

On **September 15, 2022**, Ethereum switched from **Proof of Work (PoW)** to **Proof of Stake (PoS)** consensus.

- PoS lets users vote on the next block based on how many coins they hold.
- PoW requires solving complex puzzles to validate blocks.
- This upgrade is called **Ethereum 2.0** or **The Merge**.

## Accounts and Transactions

Ethereum accounts have:

- A **20-byte address**
- A **state**

Two types of accounts:

1. **Externally Owned Accounts (EOAs):** Controlled by private keys, can send transactions. EOAs are owned by an external entity, such as a human
2. **Contract Accounts:** Controlled by contract code, can send transactions or create new contracts.

## EOA to EOA Transaction

This is the easiest type of transaction.

1. The sender makes a transaction, signs it with their private key, and sends it to the network.
2. The transaction is shared (broadcast) to the network, and miners add it to a block.
3. The transaction runs, and the blockchain's state is updated.

## Contract Accounts

These accounts include:

- Contract code (the program)
- Storage that keeps constant variables
- Ether balance (how much ETH it has)
- Nonce, which counts how many messages (transactions) have been sent from this account

**Important:** Contract accounts **do NOT** have a private key.

To create a smart contract, you need an EOA to send the transaction. After a contract is created, its code **cannot be changed**.

Creating a contract is done by sending a transaction like an EOA to EOA transaction, but:

- The **data field** contains the contract code
- The **receiver address** is empty



## Chapter 13: Non Fungible Tokens

Even though people often say "coins" and "tokens" like they mean the same thing, they are different.

- **Coins** are fungible, which means you can swap one coin for another of the same kind, one-to-one.
- **Tokens** can be fungible or **non-fungible**. Non-fungible tokens (NFTs) are unique and can't be swapped one-to-one.

This difference matters because it changes how tokens are used and valued.

**Digital tokens** are assets created on existing blockchains using smart contracts.

- **Fungible tokens** are interchangeable (like exchanging one \$20 bill for another) and divisible, just like cryptocurrencies. For example, casino chips are fungible tokens because you can exchange one chip for another of the same value.

### Non-fungible tokens (NFTs)

NFTs are unique and cannot be divided, like collectibles. They show ownership of digital or real-world things like art, music, or property. NFTs are created by smart contracts that set rules for creating, owning, and transferring them. They live on the blockchain and can be bought, sold, or traded. NFTs are popular in games, digital art, and anywhere ownership and rarity matter.

**To buy or sell NFTs**, you need a blockchain wallet that works with NFTs, like MetaMask. You connect your wallet to a marketplace on the blockchain, like OpenSea or Rarible, and look at the NFTs for sale. When you find one you want, you can either place a bid or buy it directly with cryptocurrency. After buying an NFT, you can sell it later or send it to another wallet. You don't need to know programming to create (mint), buy, or sell NFTs.

### ERC Standards

Standards are needed so developers know how tokens should behave and companies know their tokens will work with wallets and exchanges.

**ERC** stands for Ethereum Request for Comments. It is a set of rules for creating and managing tokens on Ethereum. The main ERC standards are:

- **ERC-20** for fungible tokens
- **ERC-721** for non-fungible tokens

- **ERC-1155** which can handle both fungible and non-fungible tokens, used in games and apps that need both.

Some tokens start fungible (like tickets) but become non-fungible after an event, becoming collectible.

**ICOs** (Initial Coin Offerings) help projects raise money by selling tokens to investors. These tokens usually follow the ERC-20 standard, allowing them to be traded on exchanges.

**DeFi (Decentralized Finance)** tokens are used as collateral in lending and borrowing platforms. Users deposit fungible tokens to borrow money. DeFi tokens are also used for voting and governance in decentralized groups, and as money in financial transactions.

DeFi is a new way of finance that uses blockchain to build decentralized financial services without a central authority. It includes lending, borrowing, and decentralized exchanges built on Ethereum smart contracts.

**ERC-20** is the most common standard for tokens on Ethereum. It sets rules for creating, transferring, and managing fungible tokens.

- ERC-20 tokens can be swapped one-to-one.
- The token contract keeps track of each account's balance. Balances can mean money, rights, physical items, etc.

**Allowances** let one address spend tokens from another without asking every time. This is useful in decentralized exchanges where users trade tokens without trusting the exchange.

**ERC-721**, created in 2018, is the standard for NFTs on Ethereum. It sets rules for creating and managing unique tokens that cannot be swapped one-to-one.

- Each NFT has a unique ID called **tokenId**, often a hash of its data like creator, owner, and properties.
- The data (metadata) is stored outside the blockchain but can be accessed by anyone with the token's ID.



- The pair (owner address, tokenId) is unique worldwide.

NFT contracts can include **ongoing royalties**, which automatically pay a percentage of sales to the original creator. Platforms like OpenSea manage these royalties and let creators set the percentage.

NFTs can have **unlockable content** (like a secret file or message) only accessible to the owner. This uses a function called **tokenURI()**, which gives the link to the metadata.

The **Contract metadata** is usually stored off-chain in decentralized storage like IPFS to make sure it's always available. It includes info like the token's name, symbol, and other details.

## Chapter 14: Solidity Attacks

**Solidity** is a high-level programming language that looks similar to JavaScript. It is made to work with the Ethereum Virtual Machine (EVM). Solidity is statically typed, supports features like inheritance, libraries, and custom data types. It is mainly used to write smart contracts on blockchains, especially Ethereum.

The **DAO (Decentralized Autonomous Organization)** was a smart contract made to act like a venture capital fund for Ethereum projects. It started in April 2016 and raised over \$150 million in Ether, making it the biggest crowdfunding ever at that time. The DAO let investors vote on which projects to fund and share profits from those projects.

The DAO was built as a smart contract on Ethereum using Solidity. Investors could send Ether to the contract and get DAO tokens in return. These tokens allowed them to vote on funding proposals. The contract also had a function letting investors withdraw their Ether anytime.

A **reentrancy attack** is when a hacker exploits a weakness in a smart contract by repeatedly calling a function before the first call finishes. This can let the hacker steal funds or do other harmful actions.

The DAO attack was a reentrancy attack. The hacker used the vulnerability to call the withdraw function many times before the first withdrawal was done. This let the hacker take out a lot of Ether from the contract.

To stop this attack, smart contracts should be written so an external call can't trigger the withdraw function again before the first call is completed.

### Arithmetic Overflow and Underflow

Overflow and underflow are common problems in smart contracts that cause bugs and security risks.

- **Overflow** happens when a number becomes bigger than the maximum value that the data type can hold.
- **Underflow** happens when a number goes below the smallest value the data type can hold.

Both can cause wrong results or make the contract behave in unexpected ways.

Phishing is a type of attack where a hacker tries to trick users into giving away secret info, like passwords or private keys. Phishing is common in crypto, where hackers try to steal money by fooling users into sharing their private keys or other sensitive data.



## Chapter 15: Blockchain Applications

### Supply Chain Management

Blockchain can make supply chains more clear and open, and it can also lower costs and risks. This is because blockchain gives everyone in the supply chain one shared and trusted source of information. This helps reduce arguments and mistakes. Also, blockchain helps track products better and makes people responsible for their actions, which can reduce fraud and fake products.

If a manufacturer uses a central database to track parts, equipment, locations, and service history, it can cause problems like separated data, repeated data, inconsistent data, and one place that can fail or be hacked. But a blockchain system is spread out and can't be changed easily, so it gives everyone the same true data, helping avoid mistakes and fights.

### Identity Management

SSI (Self-Sovereign Identity) is a way for people to control their own digital identities. Blockchain helps by creating decentralized IDs (DIDs) and verifiable

credentials.

- DIDs are unique IDs stored on a blockchain that prove you own your digital identity.
- Verifiable credentials are digital certificates that prove facts about you, like your age, address, or qualifications.

SSI lets people share their identity safely and privately with others.

## Chapter 16: Proof of Stake in Ethereum

Starting off recalling the Proof of Work consensus algorithm, it is important to understand that it is not the only way to reach consensus in a blockchain network.

With Bitcoin's Proof of Work, miners compete to solve a difficult math puzzle, and the first one to solve it gets to add a new block to the blockchain. This process uses a lot of energy and requires powerful computers. Because of the high cost, big mining groups called mining pools control a large part of the network's power, which means the blockchain is not fully decentralized.

It is important to know that Proof of Work is not the only way to reach consensus in a blockchain network.

Proof of Stake is another way to reach consensus that uses less energy and is less centralized than Proof of Work. In Proof of Stake, validators are picked to create new blocks based on how many coins they own. The more coins a validator has, the higher the chance they get to make the next block.

### Validators opposed to Miners

In Proof of Stake, instead of miners, there are Validators. Validators propose new blocks, but they do not solve puzzles like in Proof of Work. They also vote to decide which block will be added next to the blockchain.

One computer (node) can have many validators running on it, from zero to thousands, and they all see the same version of the blockchain.

### Being a validator

The main idea is not to solve a puzzle, but to use cryptocurrency as a stake to help confirm transactions. Validators put their own coins into the network, which they can lose if they cheat or act badly.

To become a validator, a user must deposit 32 ETH into a deposit contract and run three pieces of software: an execution client, a consensus client, and a validator client.

Validators are randomly chosen to form a group called a committee. This committee proposes and votes on new blocks. The size of the committee depends on how much ETH is staked in the network. One validator in the committee is chosen to propose the block, and the others vote on it. For a block to be added, at least two-thirds of the validators must agree (super-majority).

### Nothing at Stake

Nothing at Stake is a problem in Proof of Stake where validators might vote for multiple blocks on different forks because they don't lose anything by doing that, unlike miners in Proof of Work who must pick one chain.

To fix this, Ethereum uses slashing, which punishes validators who vote for more than one block in a fork. Validators who do this lose part of their staked coins. The same punishment applies if they try to propose more than one block in the same slot.

Slashing works because validators sign the blocks they propose. If the same validator signs multiple conflicting blocks, they can be caught and punished.

### Gasper - Casper and GHOST

Ethereum's Proof of Stake consensus algorithm is called Gasper, which combines two parts:

1. **Casper** - This chooses if a block becomes a final checkpoint. Casper makes sure that once a block is added, it cannot be changed or removed. Validators vote on blocks, and when two-thirds of them agree, the block is final.
2. **GHOST** - This is the fork choice rule that decides which branch to follow if there is a fork. It is different from the longest chain rule used in Bitcoin.

**GHOST** is a rule that looks at each participant's latest vote (Latest Message Driven). Instead of just picking the longest chain, it chooses the chain with the most weight, meaning the chain that has the most validator votes (attestations) and staked ETH supporting it.

The winning chain is the one with the most staked ETH voting for it, even if it is shorter than another chain.

**Finality** means that once a block is added to the blockchain, it cannot be undone or removed. Validators vote on blocks, and once two-thirds have agreed, the block is final.

Bitcoin does not have finality, so blocks can be replaced if a longer chain appears. But in Ethereum, once a block is finalized, it is permanent.

Validators regularly agree on recent checkpoint blocks that they will never revert, which means no forks can happen before those checkpoints.

## Chapter 17: ZeroKnowledge Proofs

In **Zero-Knowledge Proofs**, a prover can convince a verifier that a statement is true without sharing any details about the statement itself. This means the prover can prove they know a secret without showing the secret.

The "Alibaba's cave" example is often used to explain ZKPs. Imagine a cave with a door that only opens with a secret password. The prover wants to show the verifier that they know the password, but without telling what the password is. The prover opens the door to prove they know the password, but the verifier does not see the password itself.

But what if the door just opened by chance? To avoid this, the prover repeats the test many times, so the verifier becomes very sure the prover really knows the password.

### **Properties of ZKPs**

- **Completeness:** If the statement is true, the prover will always convince the verifier.

- **Soundness:** If the statement is false, the prover cannot easily trick the verifier — their luck will run out.
- **Zero-Knowledge:** The verifier learns nothing about the statement itself.

### Real World and Blockchains

Interactive ZKPs (where prover and verifier talk back and forth) are not very practical because they require both to be online at the same time and communication can be slow.

In blockchains, ZKPs help keep transfers private. Someone can prove they have the right to spend money without showing who is sending or receiving it, or the amount being sent, to the whole network.

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (ZK-SNARKs) are a special kind of ZKP. They let the prover convince the verifier that a statement is true without revealing any details, and without needing to interact multiple times.

XACML (eXtensible Access Control Markup Language) is a standard that lets administrators set rules about who can access which resources. Using ZKPs, a user can prove they have permission to access something without showing who they are or what they are accessing.