Peer to Peer - Appunti

Francesco Lorenzoni

February 2024

Contents

Part I

Introduction to P2P Systems

Chapter 1

Introduction

Opposed to Client-server architectures where there are end-hosts and dedicated-hosts (servers), in P2P Systems there only end-nodes which directly communicate with each other; they have an "on/off" behaviour, and they handle **churn**[1]. However, in P2P systems servers are still needed, but only as *bootstrap servers*, typically allowing for new nodes to easily join the P2P network.

Peers' connection in P2P is called *transient*, meaning that connections and disconnections to the network are very frequent.
Notice that since each time a peers connects to the P2P network it may have a different IP address, resources cannot be located using IP, but a different method at application layer must be used.

Definition 1.1 (P2P System) *A peer to peer system is a set of autonomous entities (peers) able to auto-organize and sharing a set of distributed resources in a computer network.*
The system exploits such resources to give a service in a complete or partial decentralized way

Definition 1.2 (P2P System - Alternative definition) *A P2P system is a distributed system defined by a set on nodes interconnected able to auto-organize and to build different topologies with the goal of sharing resources like CPU cycles, memory, bandwidth. The system is able to adapt to a continous churn of the nodes maintaining connectivity and reasonable performances without a centralized entity (like a server)*

1.1 Blockchain concepts

Definition 1.3 (Blockchain)
- ◇ *a write-only, decentralized, state machine that is maintained by untrusted actors, secured by economic incentive*
- ◇ *cannot delete data*
- ◇ *cannot be shut down or censored*
- ◇ *supports defined operations agreed upon by participants*
- ◇ *participants may not know each other (public)*
- ◇ *in actors best interest is to play by the rules*

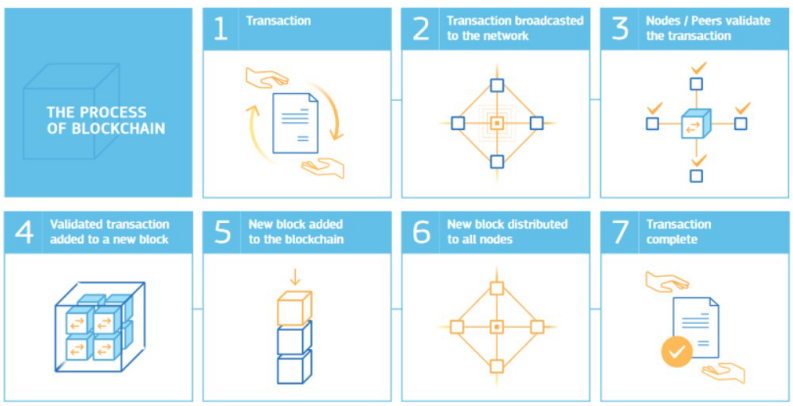


Figure 1.1: Blockchain process

[1] *"churn"* will be a recurring term. In italian it means "rimescolare"

Bitcoin were developed as an alternative way to exchange money which wouldn't need intermediaries such as banks. Today, *Ethereum* is becoming more and more popular.
NFT[2]allow to establish the owner of a digital artwork, by generating a token using a blockchain.

1.1.1 TriLemma

The Blockchain **trilemma** states that a blockchain **cannot** simultaneously provide *Decentralization*, *Security* and *Scalability*.

1.2 P2P Systems

1.2.1 Semi-Decentralized systems

An example is **Napster**, released in 2001. Napster used servers only to allow users to locate peers which could provide the desired file, delegating the actual file exchange to peers, allowing for a very few servers needed.

For the first time users are called *peers*, and the systems implemented in this way *peer-to-peer systems*

Napster had many strengths common to many P2P systems, from whose emerges the ability of peers to act both as server and a client, but also suffered from weaknesses derived from its centralization, at least for "node discovery". Napster centralized server represents a design bottleneck, and also made it target of legal attacks.

1.2.2 Fully decentralized systems

Gnutella is similar to Napster, but here no centralized server exists. Peers establish *non-transient* direct connections to search files, not to actually transfer them.

Cons
1. High network traffic
2. No structured search
3. Free-riding

1.3 P2P Overlay network

In P2P systems there is an overlay network at application level operating on top of the underlying (IP) network.

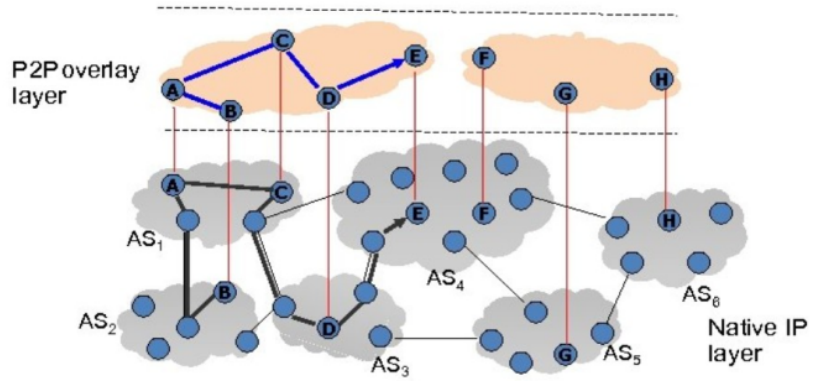


Figure 1.2: P2P Overlay networks

A P2P **protocol** —defined over the P2P overlay— defines the set of messages that the peers exchange.

1.3.1 Unstructured overlay

The two key issues here are:

- how to **bootstrap** on the network?
- how to **find content** without a central index?

Possible lookup algorithms are the following, but they all are not very scalable, and are costful in terms of performance:

- **Flooding**
- **Expanding ring**
- **Random walk**

`Gnutella`, (basic version of)`BitTorrent`, `BitCoin` are examples of unstructured overlay networks.
Boostrapping is done by querying a *well-known* DNS storing a set of "stable peers".

[2]Non Fungible Tokens

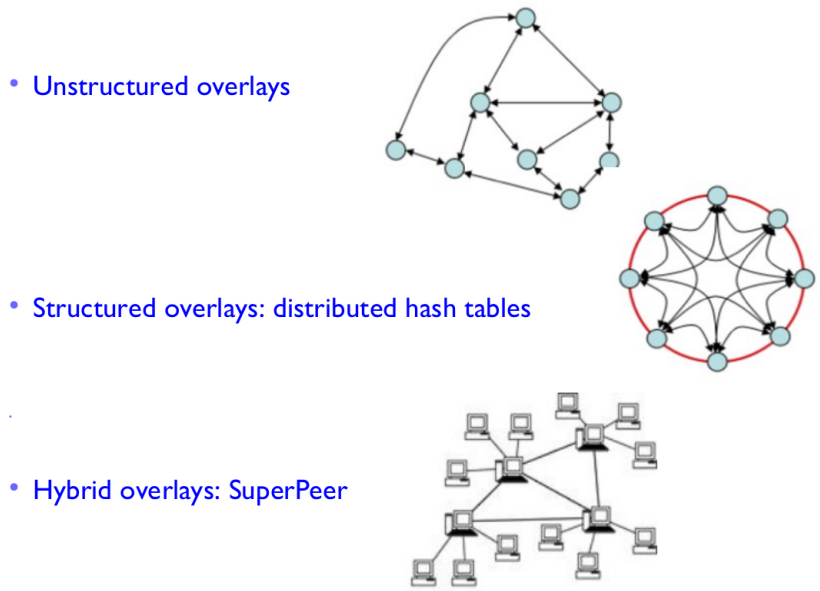


Figure 1.3: P2P Overlay Network classification

1.3.1.1 Flooding

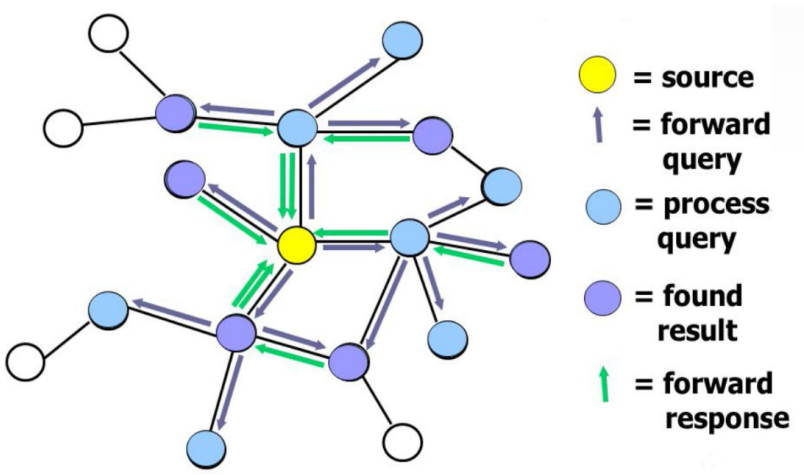


Figure 1.4: Flooding search in unstructured overlay

Messages have a *TTL* to limit the number of hops when propagating, but also a *unique identifier* to detect cycles. Once the result is found it may be transferred using a direct HTTP connection.

Flooding is not only for searching, but also to propagate transactions in the P2P network underlying a **blockchain**

1.3.1.2 Expanding Ring

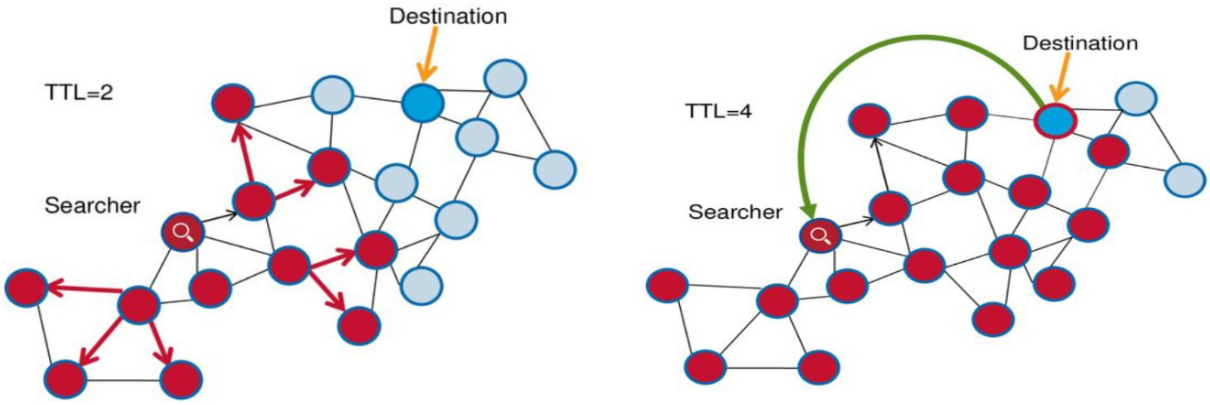


Figure 1.5: Expanding ring/Iterative Deepening
This tecnique consists in repeated flooding towards —possibly restricted to a subset of randomly selected— neighbors with an increasing TTL, implementing a BFS search.

1.3.1.3 Random walk

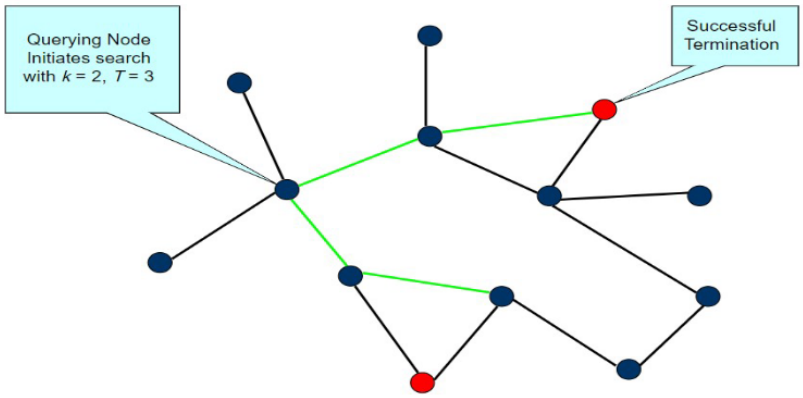


Figure 1.6: Random Walk
k indicates the number of "walkers" to be generated by the querying node. Each green path corresponds to a "walker".
A path is constructed by taking successive (single) steps in random directions defined by a *Markov Chain*, which is "memory-less". Note that only one successor node is chosen at each step.
The path is bounded by a TTL.

Random walk avoids the exponential increase in the amount of messages common in flooding, which becomes an issue for vastly populated networks.
Paths can be stopped by a TTL but also by checking periodically with the destination wether the stop condition has been met.
A querying node can also bias its walks towards high-degree nodes: higher probability to choose the highest degree neighbor.

1.3.2 Structured overlays

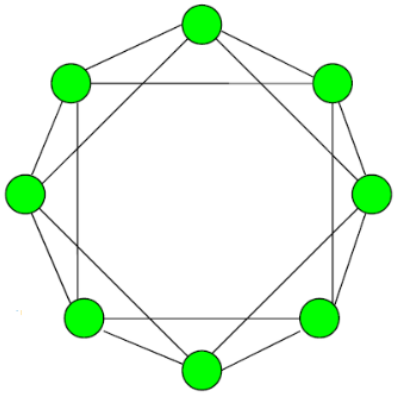


Figure 1.7: Structured overleay

The choice of the neighbours is defined according to a given criteria, resulting in a **structured** overlay network. The goal is to guarantee scalability by providing:

- ◇ *key-based lookup*
- ◇ information lookup has a given *complexity* e.g. $\mathcal{O}(logN)$

DHT based solutions, such as `Chord`, `Pastry`, `Kademlia` are examples of structured overlays.

1.3.3 Hierarchical overlays

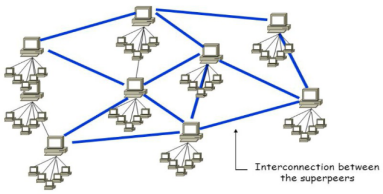


Figure 1.8: Hierarchical overlay

Peers connect to **Super-Peers** which know (*"index"*) Peer resources. The flooding is restricted to Super-Peers, but still allowing for resources to be directly exchanged between the peers.

Lookup complexity is less and the scalability is improved, but there is a lower resistance to *super-peers* churn.

In some cases —such as Gnutella— peers are "*self-promoted*" to super-peers, while in others they are statically defined.

1.3.4 Summary

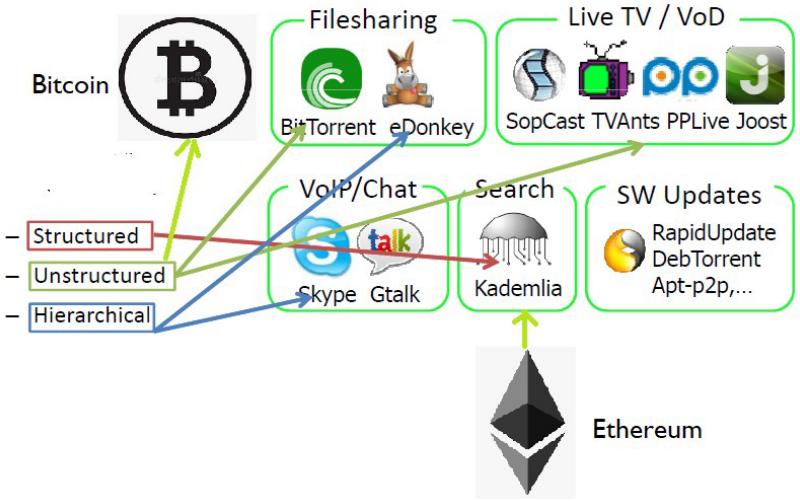


Figure 1.9: Overlay structure for known applications

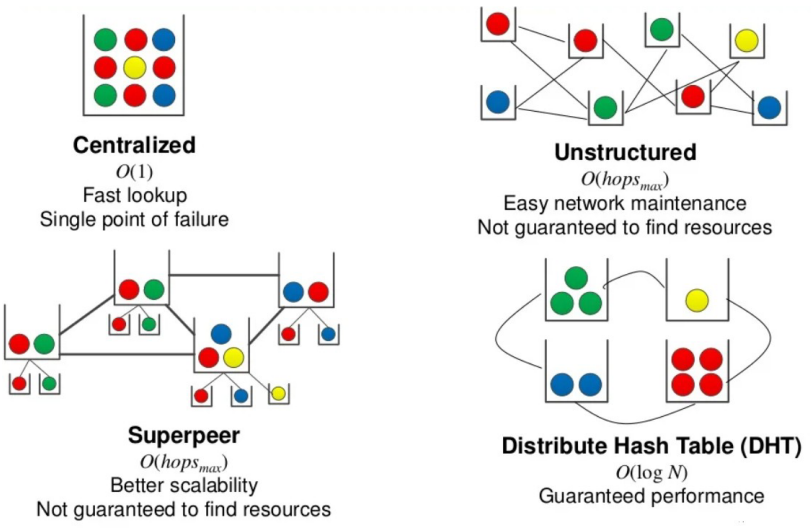


Figure 1.10: Overlays summary
DHT will be discussed in the next chapter

Chapter 2

Distributed Hash Tables

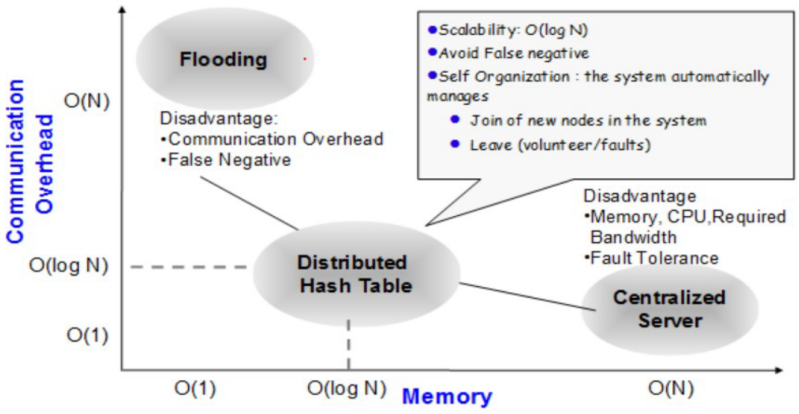


Figure 2.1: DHT Motivations

The key idea is to split the hash tables into several parts and distribute them to several servers, and to use hash of resources (or of the URLs of resources) as a key to map them to a dynamically changing set of web caches, but with each key mapped to single server; so that each machine (user) can locally compute which web cache should contain the required resource, refenced by an URL.
This technique is extended to DHT for P2P systems.

However, **rehashing** is a problem in dynamic scenarios if the hashing scheme depends directly on the number of servers: 99% of keys have to be remapped, resulting in a lot of messages exchange.

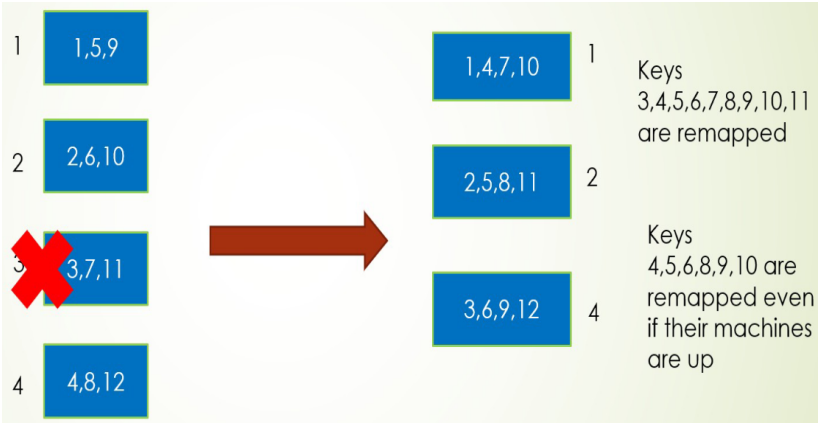


Figure 2.2: Rehashing problem

Consistent hashing is a set of hash techniques which guarantees that adding more nodes/remove nodes implies moving only a minority of data items. each node manages —instead of a set of sparse keys— an interval of consecutive hash keys, and intervals are joined/splitted when nodes join/leave the network and keys redistributed between adjacent peers.

2.1 Building CHORD DHT

- ◇ Use a logical name space, called *identifier space* consisting of identifiers $\{0, 1, 2, ..., N-1\}$
- ◇ define identifier space as a *logical ring* modulo N
- ◇ every node picks a random identifier through Hash function H.
- ◇ the function `succ(x)` returns the node with an identifier $\geq x$.
- ◇ every item v to be stored gets assigned to `succ(H(v))`

space N=16 {0,…,15}

- five nodes a, b, c, d, e
- H(a) = 6
- H(b) = 5
- H(c) = 0
- H(d) = 11
- H(e) = 2

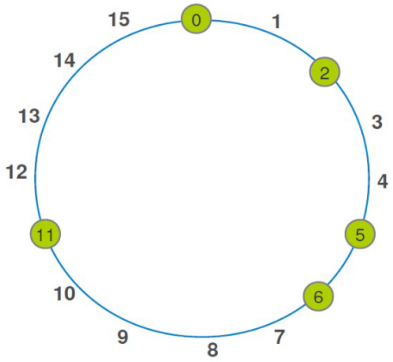


Figure 2.3: Identifier space
In this figure, the node identifiers are 16 and the green circles indicate which are the online nodes.

2.1.1 Peers joining and leaving

When a new node is **added**, we map the keys between the new node and the previous node in the hash ring to point to the new node;
those the keys will no longer be associated with their old nodes.

When a node is **removed** from the hash ring, only the keys associated with that node are rehashed and remapped rather than remapping all the keys.

In case a node suddenly disconnects from the network, all data stored on it are lost if they are not stored on other nodes; to avoid such a problem:

- ◇ introduce some redundancy (data replication)
- ◇ information loss: periodical information refresh

- the "crown" now is mapped to peer 0
- the absence of peer 11 does not affect keys belonging to other servers

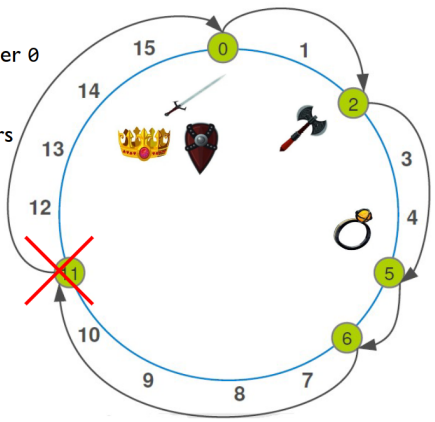


Figure 2.3: Peer 11 leaves the Network
In case a peer leaves, its keys can easily be remapped to its successor

When the hash table is **resized**, on the average,only $\frac{k}{n}$ keys need to be remapped on average, where k is the number of keys and n is the number of servers.

2.2 Finger Table and Data Lookup

- finger/routing table:
  - point to succ(n+1)
  - point to succ(n+2)
  - point to succ(n+4)
  - point to succ(n+8)
  - ...
  - point to succ(n+2^{M-1}) (M number of bits for the identifiers)
- distance always halved to the destination.

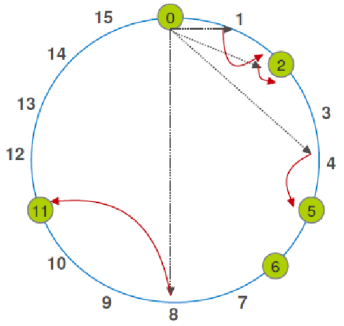


Figure 2.4: Finger table for CHORD DHT

The data lookup can be implemented by using exponential search, rather than performing a walk by asking each peer for its successor

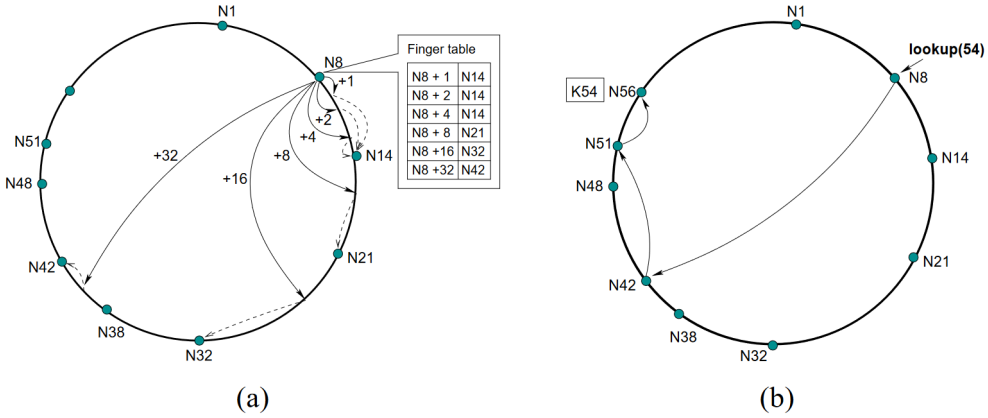


Fig. 4. (a) The fi nger table entries for node 8. (b) The path a query for key 54 starting at node 8, using the algorithm in Figure 5.

Figure 2.5: This image is a bit more precise than the previous one

Each k^{th} **finger** for the node n is defined as $finger[k] = succ(n + 2^{k-1})$, i.e. the first node on the circle that succeeds $n + 2^{k-1}$ mod 2^m, having $1 \leq k \leq m$.

The lookup algorithm, instead of walking through successors to find the desired key, can be sped up by using the **finger table**. Here is the algorithm, which is logarithmic in the number of nodes. The use of finger tables not only allows to speed up the search, but also to reduce the number of nodes each peer has to keep track of.

In Fig. 2.5 node 8 the successor of 54 he knows of is 1, which does not contain the key 54, so then 8 finds the closest preceding node to 54 in its finger table, which is 42, which finds again as successor of 54 the node 1, since he knows only $succ(42 + 8) = 51$ and $succ(42 + 16) = 1$, so asks 51 (closest preceding node) to find the successor of 54, which is 56 (finally ☺).

```
// ask node n to find the successor of id
n.find successor(id)
   if (id ∈ (n, successor])
      return successor;
   else
      n' = closest preceding node(id );
      return n'.find successor(id);
// search the local table for the highest
    predecessor of
    id
n.closest preceding node(id)
   for i = m downto 1
      if (finger[i] ∈ (n, id))
         return finger[i];
   return n;
```

So, **Data Lookup** is performed by computing the hash $h(x)$ of the searched object, asking for $succ(h(x))$ whether they have x, and if not, propagating the query to farthest node[1] which has an identifier smaller than $h(x)$, which then recursively applies the same algorithm, until the object x is found.

[1] Which is found using the finger table

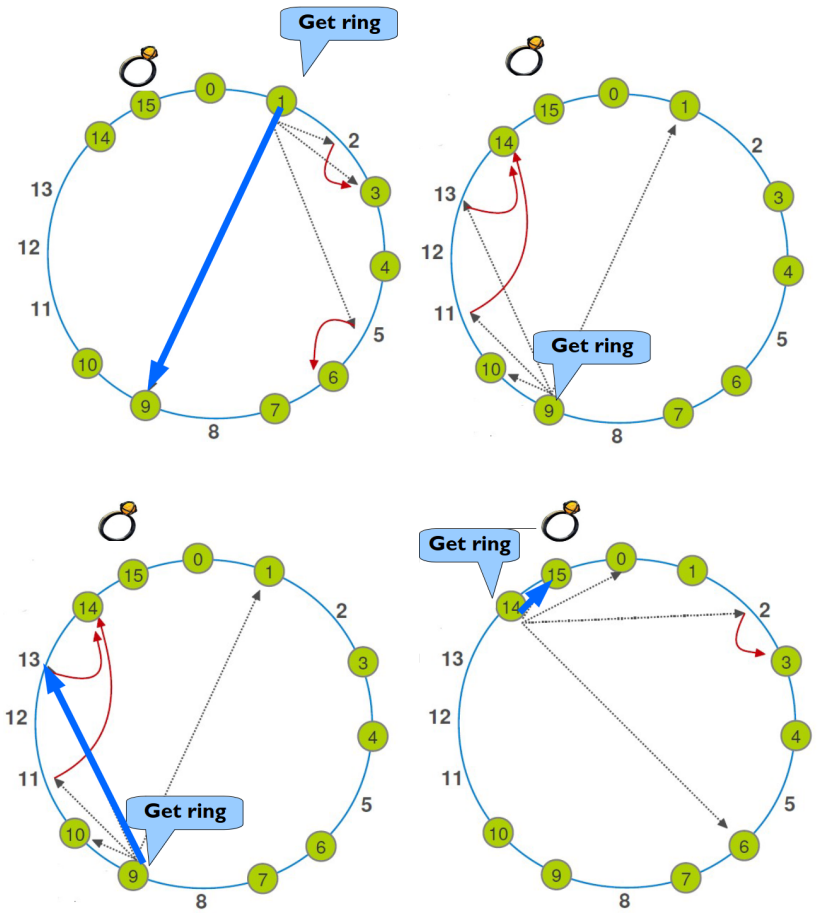


Figure 2.6: Lookup performed in the CHORD DHT
Dotted arrows here indicate the nodes each peer knows of, and the big solid arrows indicate the path the query follows to find the desired key.

2.2.1 Addressing data

Data was usually addressed by **location**, a `http://` link to locate resources; Such link is an identifier that points to a particular location on the web.
This approach forces us all to pretend that the data are in only one location, which is not the case for distributed storage.

IPFS instead uses **content addressing**, which exploits the cryptographic hash of the content to identify it.

2.2.2 API, Lookup and Various Properties

To avoid having a node managing a bigger portion of the identifier space, a uniform hash function may be used.
Most DHT provide a simple inferface `PUT,GET,Value`, usually *without* the possibility to move keys.

Approach	Memory for each node	Communication Overhead	Complex Queries	False Negatives	Robustness
Central Server	O(N)	O(1)	✓	✓	✗
Pure P2P (flooding)	O(1)	$O(N^2)$	✓	✗	✓
DHT	O(log N)	O(log N)	✗	✓	✓

Figure 2.7: Lookup time complexity comparison

DHT

- ◇ Routing is based on key (unique identifier)
- ◇ Key are uniformly distributed to the DHT nodes
  1. Bottleneck avoidance
  2. Incremental insertion of the keys
  3. Fault tolerance
- ◇ Auto organizing system
- ◇ Simplex and efficient organization
- ◇ The terms “Structured Peer-to-Peer“ and “DHT“ are often used as synonyms

Chapter 3

Kademlia

Kademlia is a protocol used by some of the largest public DHTs
- ◇ BitTorrent Mainline DHT
- ◇ Ethereum P2P network
- ◇ IPFS

It has three key charateristics which are not offered by other DHTs
1. routing information spreads automatically as a side-effect of lookups
   The list of known nodes is updated with the most recently contacted ones
2. flexibility to send multiple requests in parallel to speed up lookups by avoiding timeout delays (parallel routing)
   Sending requests to multiple nodes in the same k-bucket
3. iterative routing
   At each routing step of the query, the queried node sends a report to the starting querying node, even if it could not answer the query.

3.1 Structure

Kademlia exploits the leaves of a complete binary **Trie**[1] to define the logical identifier space;

Note that not all leaves correspond to nodes (peers). In Fig. 3.1, only the **circled** leaves are nodes.

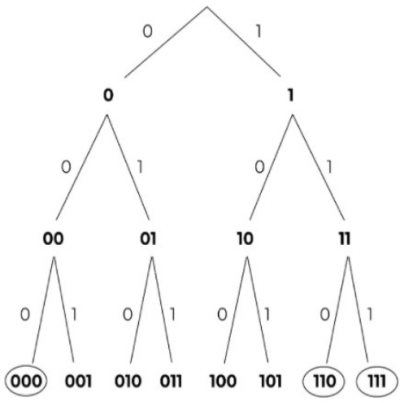


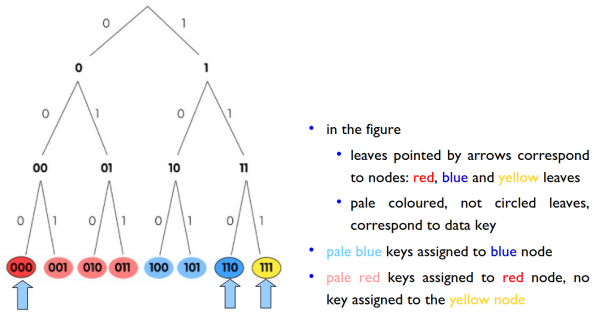
Figure 3.1: Trie

[1] k-ary search tree and prefix tree

3.1.1 Assigning keys to leaves

The rule to partition the keys (content) among the nodes must respect the rules of *consistent hashing*.

Definition 3.1 (Partitioning rule) *A key is assigned to the node with the "lowest common ancestor": Find the longest prefix between the key and the node identifier, and then assign the key to such node.*



3.2 Distance - XOR Metric

In Fig. 3.1.1 the assignment of keys to blue nodes instead of yellow ones is arbitrary, since the two nodes "tie" for the longest common prefix challenge, leading to no keys being assigned to the yellow nodes.
The problem may be solved by assigning keys to the node with the *"closest"* identifier to the key.

*How to compute the **distance** between two nodes?*

To solve this problem, Kademlia uses the **XOR metric**. The XOR metric consists in computing the XOR between a key and a node ID, and interpreting the result as an integer to get the distance.
This ensures that there is a unique closest node to a key, and that the distance between two nodes is the same in both directions.

3.3 Routing Table

In order to look for data, Kademlia's key idea is to store a logarithmic number of node IDs and their corresponding IP addresses and some contact taken from the identifier trie.

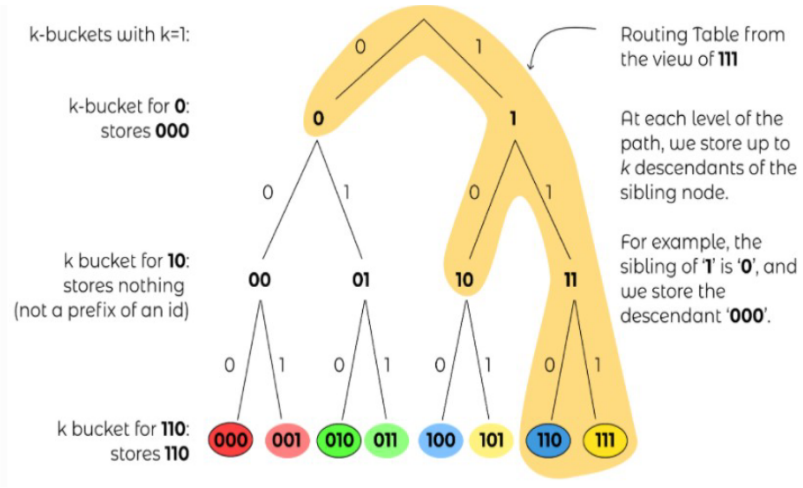


Figure 3.1: Neighbours and buckets

Each k-bucket corresponds to a prefix and covers a subset of the identifier space: the set of all k-buckets covers the whole identifier space which 2^m, with m[2] the length of an identifier; more precisely the i-th bucket covers the range $[2^i, 2^{i+1})$, so as i increases, so does the range of the bucket, exponenetially.
The first entries of the routing table correspond to peers sharing a long prefix with the owner of the routing table; the last entries instead of the routing table correspond to peers sharing a smaller prefix, and cover a larger set of identifiers. The value of k is defined such that the probability that a crash of more of k nodes is a rare event. Nodes in each bucket are maintained ordered such that *least recently contacted nodes are in the first positions of the list*.

[2] $m = 160$ in Kademlia

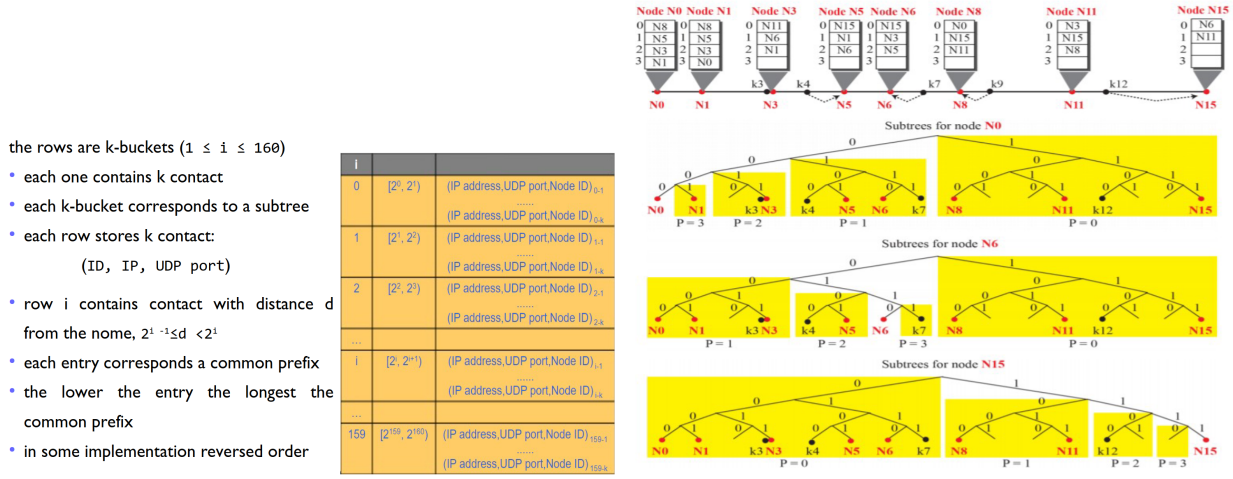


Figure 3.2: Deeper into Routing Tables and buckets

3.3.1 Routing Strategies

In **iterative routing** the node n sending the lookup request manages the whole search process. At each routing step n receives a reply including a notification of the next routing step.

In **recursive routing** the node n sending the lookup request delegates the search process to the node replying to the request.

Parallel routing is a strategy that allows the requester to send multiple requests in parallel to speed up the lookup process.

3.4 Key Lookup

The idea for key lookup is:

1. Find the closest node to the key in your routing table, computing the XOR distance in case of ties
2. While the closest node you know of does not have the key and has not already responded
   i. Ask the closest node you know of, for the key or a closer node
   ii. If the closest node responds with a closer node, update your closest nodes set. (i.e. current bucket)

At each iteration, the XOR metric is reduced by 1/2 and results in smaller size k-buckets

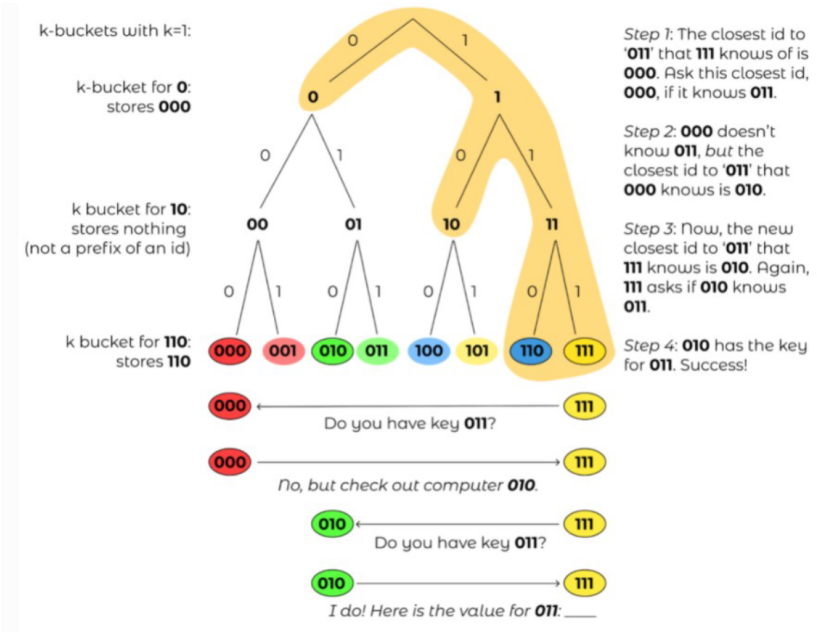


Figure 3.3: Kademlia Lookup at a glance

3.5 Protocol Messages

1. `FIND_NODE` $v \longrightarrow w$ (T) (v,w nodes, T target of the look up)
   the recipient of the message (w) returns k (IP address, UDP port, Node ID) triples for the k nodes it knows about closest to the target T.
   these triples can come from a single k-bucket, or they may come from multiple k-buckets if the closest k-bucket is not full.
   in any case, the recipient must return k items, unless there are fewer than k nodes in all its k-buckets combined, in which case it returns every node it knows about
2. `FIND_VALUE` $v \longrightarrow w(T)(v, w\ nodes, T\ value\ looked\ up)$
   in: T, 160-bit ID representing a value
   out: if a value corresponding to T is present in the queried node (w), the associated data is returned otherwise it is equivalent to `FIND_NODE` and w returns a set of k triples
   If `FIND_VALUE` returns a list of other peers, it is up to the requester to continue searching for the desired value from that list
3. `PING` $v \longrightarrow w$
   probe node w to see if its online
4. `STORE` $v \longrightarrow w$ $(Key, Value)$
   instructs node w to store a $\langle key, value \rangle$ pair
   the node has been retrieved through a

The actual **Lookup algorithm** is based on `FIND_NODE`. many `FIND_NODE` can be executed in parallel, according to α that is a system-wide concurrency parameter.

With $\alpha = 1$, the lookup algorithm is similar to *Chord*, one step progress each time

Lookup procedure is the same for `FIND_VALUE` and `FIND_NODE`

k-closest= α contacts from the non-empty k-bucket closest to the key
if there are fewer than α contacts in that bucket **then**
    k-closest = *k-closest* U closest contacts from other buckets.
closestNode = the closest node in k-closest
/* recursive step
repeat
    select from *k-closest*, α closest contacts which have not been queried yet
    send in parallel, asynchronously `FIND_NODE` to the delected contacts
      each contact, if live, returns k nodes
    add to *k-closest* the new received nodes and update *closestNode*
until no node closer to the target than *closestNode* is returned

send in parallel asynchronously `FIND_NODE` to the k closest nodes it has not already queried

return the k closest nodes

Figure 3.4: Kademlia Lookup Algorithm

Part II

BitTorrent and Blockchains

Chapter 4

BitTorrent

The goal of *Content Distribution Networks* is to distribute web contents to hundreds of thousands or millions of simultaneous users, exploiting data and/or service **replication** on different **mirror servers**.

In **P2P CDN** the initial file request are served by a centralized server, and further requests served by peers which have already received and replicated the files (***seeders***), without involving the initial server.

BitTorrent in a nutshell

- ◇ Basically a *Content Distribution Network* (`CDN`)
- ◇ A distributed set of hosts cooperating to distribute large data set to end users.
- ◇ Efficient content distribution systems using *file swarming*
- ◇ Does *not* perform all the functions of a typical P2P system, like searching
- ◇ Rather than providing a search protocol itself, was designed to integrate seamlessly with the Web and made file descriptors available via Web, which could be searched with standard Web search
- ◇ *File swarming*: a peer makes whatever portion of the file that is downloaded immediately available for sharing

4.1 Deeper into BitTorrent

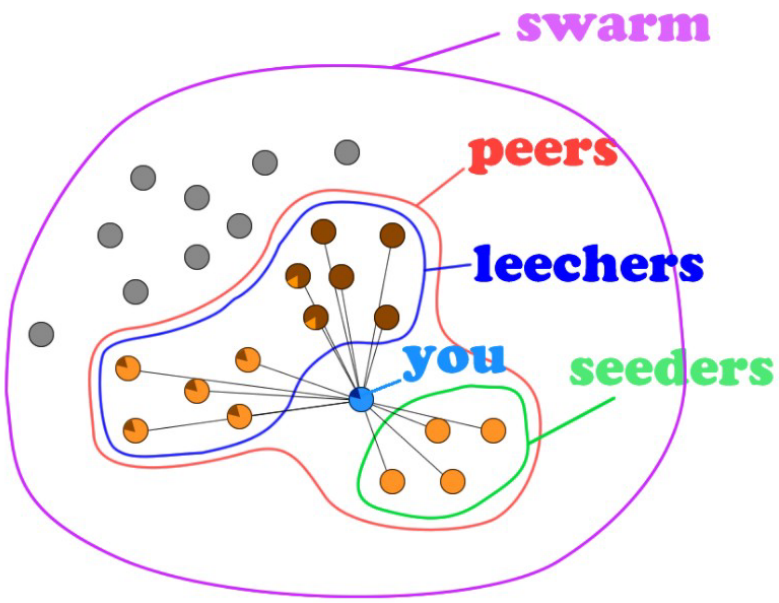


Figure 4.1: Swarm schema

4.1.1 Glossary

- **tracker**: active entity which coordinates the peers sharing the file, taking trace of who is currently providing the content
  - Joe connects to the tracker announcing the content
  - the tracker now knows Joe is providing the file
- `.torrent` a descriptor of the file to be published on a server, which includes a reference to a tracker
- **swarm** set of peers collaborating to the distribution of the same file coordinated by the same tracker
- **seeder** peer which owns all the parts of the file
- **leecher** peer which has some part or no part of the file and downloads the file from the seeders and/or from other lechers.

4.1.2 Protocol Overview

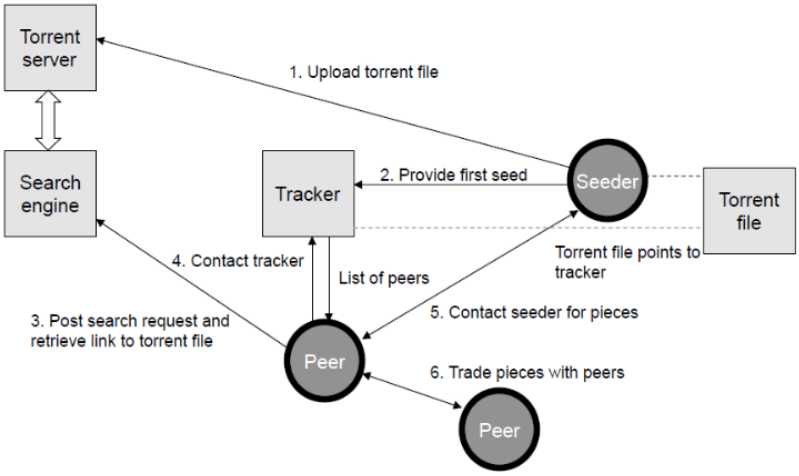


Figure 4.2: BitTorrent protocol overview
BitTorrent protocol is built on top of HTTP

Seeder

1. Upload the .torrent on a Torrent Server, indicating a Tracker address and the infohash of the file
2. Opens a connection to the Tracker and informs it of its own existence: for the moment, it is the only peer which owns the file

Peers

3. Retrieves the file descriptor (.torrent) and opens it through the BitTorrent client
4. Opens a connection to the tracker and informs it of its own existence and receives from the tracker a list of peers of the swarm
   The tracker's address Is included in the `.torrent` file
5. Opens a set of connections with other peers of the swarm.

Objects are serialized in **Bencode**, which is —not popular as `JSON`— used only in torrent; provides 4 data types: String, Integer, Lists and Dictionaries.
Content is split into chunks called pieces (256KB - 2MB): when a peer receives a piece, it becomes the seeder of that piece.
There is a SHA-1 hash per piece stored in the .torrent file, used to check the piece once it is fully downloaded, allowing to require retransmission in case the check fails.
Pieces size got adapted to have a reasonably small .torrent file

Pieces are then split in **subpieces** (***blocks***) of 16KB, with each one downloadable from a different peer, optimizing the bandwith and allowing *pipelining*, decreasing the overall download time. Pieces are the smaller units of retransmission, and are checked using a SHA-1 hash once all of its blocks are downloaded.

Trackers keep a database of swarms identified by torrent hash, and knows also the state of each peer in each swarm. In the last versions, **trackerless** BitTorrent uses *Kademlia DHT* to avoid the centralization point of the tracker.

4.2 Pieces selection

Figure 4.3: BitTorrent policies over time

The order in which pieces are selected by different peers is critical for good performance, to avoid making peers end up stuck with the same pieces.

Policies

- ◇ **Strict Priority**
  Complete the "assembling" of a piece before asking for another piece.
- ◇ **Rarest First**
  Download the rarest pieces first. Allowed since the peers have a local view of the availability of each piece. Acquiring rare pieces means that many peers will be interested in them, increasing the download speed for the peer owning them ("tit-for-tat" approach).
- ◇ **Random First Piece**
  Choose a random piece —only— in the bootstrap phase. Used because peers need to acquire a piece ASAP to start negotiating with other peers, i.e. downloading remaining pieces
- ◇ **Endgame**
  When the file download is almost terminated, the remaining pieces are required in *parallel* to all peers who own them. This policy is executed for a small period of time. It is used because typically most downloads slow down when the file is almost complete.

4.2.1 Free Riders

Free riders in BitTorrent are peers that do not put their bandwidth at disposal of the community.
Several non official BitTorrent clients enable the user to limit the upload bandwidth as they like. The performance of the whole network relies on the cooperative behavior of the peers, making free riders a problem.

An approach to address this issue is based on **reciprocity**, allowing a client to obtain a good service if and only if it gives a good service to the community, by exploiting a dynamic strategy based on connection monitoring called "Tit for Tat", implemented using **choking**:
"choking" means *temporarily* refusing to upload to another peer, but still downloading from them; the principle is to "upload to peers who have uploaded to us", so each peer, periodically, evaluates for each neighbor the *upload rate* and the *download rate*, and decides a fixed number of neighbors to *choke* (and *unchoke*)

Choking

The local peer can receive data from a remote peer if

- ◇ The local peer is *interested* in the remote peer
- ◇ The remote peer *unchoked* the local peer

Choking only peers that upload the most to the local peers would lead to ignoring peers that recently join the network and to the lack of discovery of connections actually better than the used ones.
To avoid this, BitTorrent uses **optimistic unchoking**, i.e. one random peer is being unchoked.
Then, every 30s an interested and choked peer is selected at random **planned optimistic unchoke** (`POU`), and if this new connection turns out to be better than one of the existing unchoked connections, it will replace it.

In case a peer is choked by everyone, it follows an **anti-snubbing** policy, by increasing the number of simultaneous optimistic unchocke to more than one.

For *seeders* this schema does clearly not apply, since they do not have to download anything; hence they use a different choking algorithm: unchoke peers with the highest upload rate, ensuring that pieces get uploaded and replicated faster.

4.3 DHT and BitTorrent

Even though a tracker requires very little resources, it is still a single point of failure, and a potential bottleneck for the system, besides it may be target of several[1] attacks.

To address this problem, Bittorrent Inc. introduces its own DHT, called *Mainline DHT*, based on Kademlia, but with some improvements concerning

- ◇ Routing table management
- ◇ Look-up

The main purpose of Mainline DHT is to provide a "trackerless" peer discovery mechanism to locate peers belonging to a swarm.

The DHT acts as a distributed tracker, and stores the content's infohash (*key*) and the list of peers in the swarm (*value*). Each node implements both the DHT and the BitTorrent protocol.

The protocol provides four messages:

1. `PING` to check if a node is available
2. `FIND_PEER(target_id)` to find the node closest to a given ID
3. `ANNOUNCE_PEER` to announce that a peer is part of a swarm
4. `GET_PEERS(infohash)` to find the peers in a swarm

[1] Not only cyber, also *legal* ones

The DHT is exploited just as a regular DHT, where the value to seek is the list of seeders for a given infohash. `Alice` asks its neighbors if they know the seeders for an infohash (tracker info for the infohash), and if they do not, they ask their neighbors, and so on; When `Alice` receives from node `15` the info that node `4` is a seeder for the infohash, they will connect to node `4` to download the file.

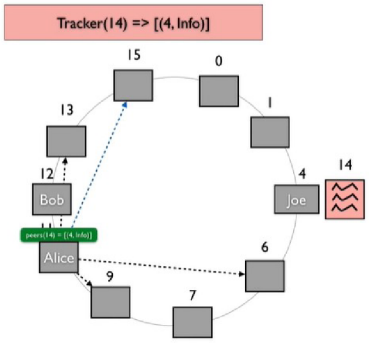


Figure 4.4: `15` has tracking information for the pink (`14`) content, so it returns info about all the peers providing the file: in this case, only `Joe` with `ID 4`

Chapter 5

Blockchain

The basic concepts concerning Blockchains are

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

Each **block** is made up of *Data*, *Hash* and the *Hash of the previous block*; pointers to the previous block are used to ensure the **order** of the blocks, resulting in a **chain** of blocks.

Tamper freeness refers to changing one hash causes changing the hash of the following blocks, implying not only to recompute some hashes, but also to find a value that combined with the new hash solves the *Proof of Work*.

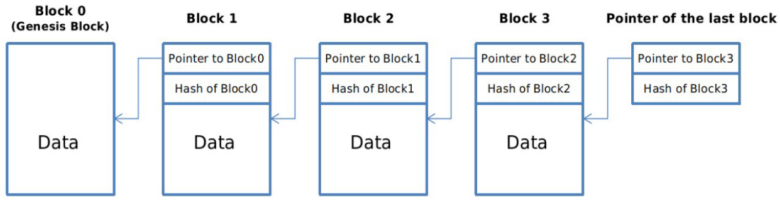


Figure 5.1: Hash pointers and hash of the block prevent attackers from tampering with the blockchain

A **ledger**[1] acts like a notary, and is replicated on each node of a P2P network, it is immutable and benefits of the tamper freeness property.
The ledger is like a bullettin storing operations and their order. It must be an **append-only** list of events, and also **tamper-proof**.

If a ledger is organized as a list of blocks, we call it a **blockchain**.

5.1 Consesus and challenges

Consensus is the mechanism which defines who decides which operation will be added to the blockchain, and which operation among those to be confirmed will be added. Consesus is implemented by *voting*, but there a few things to handle to avoid double spending and fake votes. *Sybil* attacks are a common issue in voting systems, where a single node can fake multiple identities to gain more voting power.

The two main challenges for the ledger are keeping consistency in case of network jitter and possible delays, and avoid nodes to fake results. An idea is to establish *consensus* using a **Proof of Work**, which requires the voting system to be hardly fakeable, i.e. resolving a difficult computational problem. Enforcing a PoW is a way to avoid the Sybil attacks issue, as it implies to have a lot of computational power to fake a vote.
The Proof of Work is often compared to a lottery, where tickets for the lottery are very expensive (computational power), and the winner is the one who solves the problem first, and gets the right to decide which block to add to the blockchain.

[1] *"Libro mastro"* in italiano

In case multiple winners are picked at the same time, a **fork** is created, and the network must decide which fork to follow. The longest fork is usually the one to be followed, as it is the one with the most computational power behind it.

5.2 Restricted access

It is possible to build a permissioned blockchain, where only a restricted set of nodes can vote and add blocks to the blockchain. This is useful in case of a consortium of companies, where the blockchain is used to store transactions between them.

The blockchain may be exploited to demonstrate the validity of the supply chain of a product, or to store the history of a product, from the raw materials to the final product, allowing also to determine which point in the supply chain a product has been compromised.

Chapter 6

Tools for DHT and Blockchains

6.1 Cryptographic Tools

Definition 6.1 (Hash Function) *An hash function converts a binary string of arbitrary length to a binary string of fixed length*

6.1.1 Hash functions and collisions

Non-crypto hash functions have low collision probability, but for an adversary specifically looking to produce one, it may be easy to succeed.
For example, the *Cyclic Redundancy Check* (`CRC`) —which essentially is the remainder in a long division calculation— was long mistakenly used where instead crypto integrity was required. Even if it is unlinkely to generate a collision using random errors, it is reasonably easy for an adversary to find one.

Note that collisions *always* exist, because the codomain is always smaller than the domain of the function.

The **pigeonhole principle** states that if n pigeons are put into m pigeonholes, with $n > m$, then at least one pigeonhole must contain two or more pigeons.

The term **hash security** refers to how hard is to *find* a collision for a given hash function.

Birthday Paradox

A hash function H has n possible output bits, so 2^n possible output strings.
If H is applied to k inputs, in order to have the probability of a collision greater than $1/2$, k must be greater than $\sqrt{2^n} = 2^{n/2}$. This is known as the **birthday paradox**.

6.1.2 Cryptographic Hash functions

Two main properties must hold for an HF to be cryptographic:
1. **Adversarial collision resistance**
2. **One way function**

These are formalized as:
1. *Pre-image* resistance
   $\forall y \in Y.$ *hard to find* $x \in X \mid h(x) = y$
   "one-way function"
2. *Second pre-image* resistance *given* $x \in X, y = h(x)$. *hard to find* $x' \in X \mid h(x') = y$
   Also called *weak collision resistance*
3. *Collision* resistance *Hard to find* $x_1, x_2 \in X. x_1 \neq x_2 \wedge h(x_1) = h(x_2)$
   Also called *strong collision resistance*
   Given a $m - bit$ hash function, the attacker needs about $2^{m/2}$ brute force computation to find a collision, as stated by the birthday paradox.

6.1.3 Hiding and Puzzles

For cryptocurrencies and blockchains also **hiding** and **puzzle-friendliness** are required.

Definition 6.2 (Hiding) *a hash function H is said to be hiding when a secret value R is chosen from a probability distribution that has high min-entropy, then, given $H(R||x)$, it is infeasible to find x*

Commit and Reveal

1. Alice commits to a value x by sending $H(x)$ to Bob
2. Alice reveals x to Bob
3. Bob checks that $H(x)$ matches the commitment

Basically the commitment is a *hash* of the value, and the value is revealed after the hash is sent. Suppose you have to remotely play Rock-Paper-Scissors: you can both commit to your choice by sending the hash of it, and then reveal them at the same time, resulting in a fair game.

A hash/search puzzle consists of:
- ◇ Cryptographic hash function, H
- ◇ Random value, r
- ◇ Target set, S
- ◇ Solution of the puzzle is a value x, such that: $m = r||x \wedge H(m) \in S$

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

Definition 6.3 (Puzzle friendliness) *H is said to be puzzle friendly if:*

- ◇ *For every possible n-bit output value y, if r is chosen from a distribution with high min entropy, then it is infeasible to find x such that $H(r||x) = y$ in time significantly less than 2^n.*

Puzzle-friendly property implies that *no* solving strategy to solve a search puzzle is much better than *trying exaustively* all the values $x \in X$.

6.1.4 Use cases

- ◇ **Data fingerprinting**
  In general $H(x) = H(y) \Rightarrow x = y$, so H allows us to avoid comparing the whole files
- ◇ **Message Integrity**
  $H(x)$ may be used as a checksum value
- ◇ **DHT**s
- ◇ **Digital Signature**
  Hash functions are widely used for public-key aymmetric algorithms, for ensuring both *confidentaliaty* and message *integrity* (and *authentication*), by appending a **digest** to the message.
  Recall that without a *digital certificate* proving the identity of the sender, only "weak authentication" is provided; without one, a third party may impersonate someone else.
  The major challenge for digital signatures is to prevent adversaries from learning how to sign messages by analysing the verification-key.

6.2 Data Structures

6.2.1 Bloom Filters

Bloom Filters answers queries like *"is k an element of S"*; they assess the *Set Membership* problem. Bloom filters are fast and lightweight but provide a proabilistic answer

$$BF(k) = \begin{cases} 0 & k \notin S \\ 1 & k \text{ may be in } S \end{cases} \tag{6.1}$$

Given n elements mapped on m bits through k hash functions, the probability of false positives is

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

A common use of BFs is to perform the **intersection** between them. It is possible to compute also the **union** of two BFs, by computing the Bitwise OR of the two BFs. **Delete** operation is not possible, but it is possible to use a *Counting Bloom Filter* to keep track of the number of times an element has been inserted, and increment or decrement the counter accordingly.

Bloom Filters are used by Ethereum, Google, and Bitcoin.

6.2.2 Merkle Hash Tree

It is a data structure summarizing a big quantity of data, with the goal of verifying the correctness of the content. A **Merkle Hash Tree** consists of a complete binary tree of hashes built starting from an initial set of data:

- i^{th} leaf stores the hash h_i of f_i
- An internal node contains the concatenation of the hashes of the sons of the node
- The last hash stored in the root is called *Merkle Root Hash*

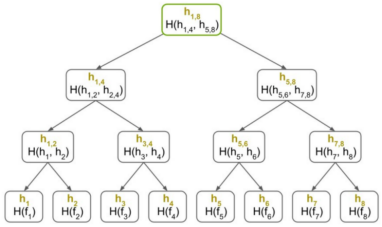


Figure 6.1: Merkle

A collision-resistant hash function *Merkle Hash Tree* (`MHT`) takes n inputs $(x_1, \ldots, x_n)$ and outputs a Merkle root hash $h = MHT(x_1, \ldots, x_n)$. Such function has an important property:

Imagine Alice (*verifier*) knows only the Merkle root hash h;
Bob (*prover*) can give Alice one of the values x_i and convince Alice that it was the i^{th} input used to compute h. To convince her, Bob gives Alice an associated Merkle proof without showing all the other inputs; if a Merkle proof says that x_i was the i^{th} input used to computed h, no attacker can come up with another Merkle proof that says a different $x_i' \neq x_i$ was the i^{th} input uses in MHT

Definition 6.4 (Merkle Proof Consistency Theorem) *It is unfeasible to output a Merkle root h and two inconsistent proofs π_i and π_i' for two different inputs x_i and x_i' at the i^{th} leaf in the tree of size n*

This can be proved by intuition as follows: if the proof verification had yielded the same hash but with a different ~~file~~ leaf $f_i' \neq f_i$ as the i^{th} input, this would yield a collision in the underlying hash function H used to build the tree; but such a collision is not possible if H is collision resistant.

The reason to build a Merkle tree is to avoid sending the whole tree to the verifier, but only the path from the leaf to the root, which is logarithmic ($logn$) in the number of leaves. I case of thousands of leaves, the proof is much shorter than the tree itself.

Enlightening Example - Cloud File Integrity

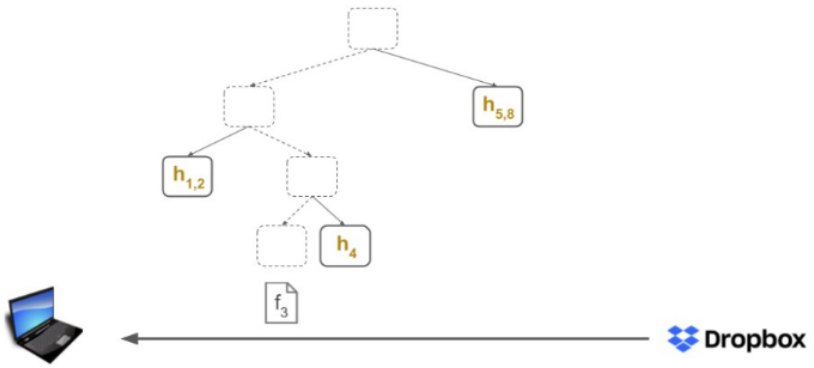


Figure 6.1: Cloud file integrity use case

Suppose that the user downloads the file f3 —which was earlier on stored on the user's PC— and wants to check that Dropbox hasn't tampered/corrupted it. The user must keep only the **hash root** of the file they had uploaded on Dropbox, not the whole Merkle Tree.
Dropbox can provide —along with the file— a portion of the original Merkle Tree (the *Merkle Proof* or *Membership proof*), only the nodes needed for the user to compute the Merkle proof, i.e. computing the sequence of hashes "filling the blanks" up to the root and check that the resulting root matches the one stored on the user's PC.

I think that the Dropbox cannot fake a Merkle tree by choosing fake $h_4, h_{1,2}, h_{5,8}$ such that the root is the one expected by the user, due to the *collision resistant* property of H

6.3 Tries and Patricia Tries

Trie

- ◇ The root node stores nothing.
- ◇ Edges are labeled with letters and a path from the root to the node represents a string.
- ◇ The nodes come with an indicator, which indicates whether that node represents the end of a string.

This is very space consuming since every node stores a label. The trie may be compressed by storing only the first different prefixes[1] in the nodes, resulting in an equivalent **Patricia Trie**, shown in the second figure.

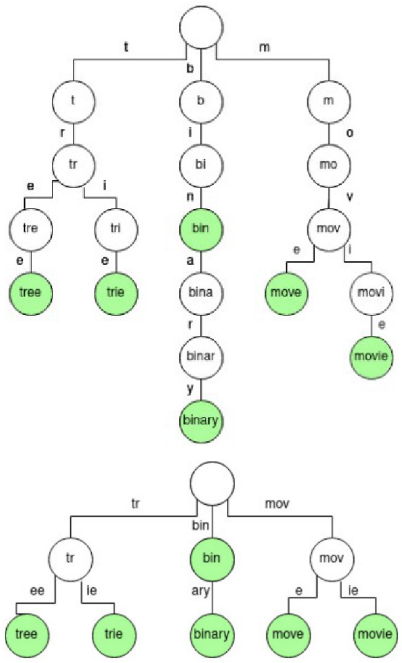
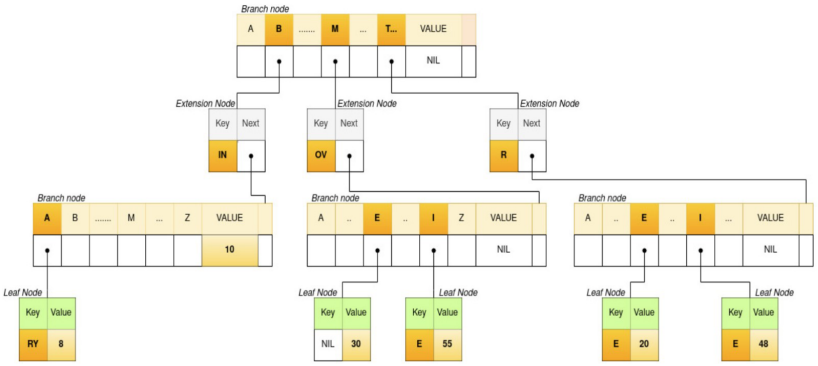


Figure 6.2: Trie and corresponding Patricia Trie

6.3.1 Patricia Merkle Trie



Key-value mapping represented in the Patricia trie

```
(BIN, 10) (BINARY, 8) (MOVE, 30) (MOVIE, 55) (TREE, 20) (TRIE, 48)
```

Figure 6.2: Patricia Merkle *key-value* Trie

A **Patricia Merkle Trie** is a combination of a Merkle Tree and a Patricia Trie, introduced by Ethereum. Nodes may be of three types:

1. *Branch* Node - multiple children, and may store a **value** referred to the key which points to such Branch Node (if it is indeed a key and not a prefix)
2. *Extension* Node - single child, and stores the **prefix** of the key
3. *Leaf* Node - stores the final bits of the **key** and the **value**

A trie can be used to store a set of key-value pairs, where the key is a string and the value is either a pointer to another child (or node), or the value itself, in case of a leaf node.

In Ethereum the Patricia Merkle Trie is used to store the state of the blockchain, where the key is the address of

1. an *account* and the value is the account *balance*.
2. a *transaction* and the value is the *transferred amount*.

Since the nodes of such tree are multi-element, and hash functions are instead applied to strings, some sort of **serialization** is applied:

- ◇ `HP` - `Hex Prefix Encoding` for the keys
- ◇ `RLP` - `Recursive Length Prefix Encoding` for the values (which may be *structured data*)

[1] Actually also only the first different character should be ok, if I recall corecctly from Algorithm Engineering course

Part III

Bitcoin

Chapter 7

Bitcoin Transactions and Scripts

Before Bitcoin in the late 90s there was an idea for a digital currency called `e-Cash`. `e-Cash` introduced the concept of **blind signatures**, which allowed a bank to sign a transaction without knowing the details of the transaction. The concept was to go a step further than plain public key cryptography, by adding a **nonce**[1] to be —mathematically— combined with the data in transit, obtaining *"scrambled data"*. The bank would *"blind sign"* the scrambled data, without being able to know the original data.

7.1 Bitcoin release

Later on, in 2008, Satoshi Nakamoto[2] introduced Bitcoin, a decentralized digital currency, with some key properties:

- ◇ *Double-spending* is prevented with a peer-to-peer network.
- ◇ No mint or other trusted parties.
- ◇ Participants can be *anonymous*.
- ◇ New coins are made from Hashcash style *proof-of-work*.
- ◇ The *proof-of-work* for new coin generation also powers the network to prevent *double-spending*

Bitcoin differs from `e-Cash` in that it is a decentralized and *unstructured* entirely P2P system, with no central authority such as banks.

7.1.1 Unhappy episodes

In 2014 Mt GOX, a Bitcoin exchange, filed for bankruptcy after losing 850,000 Bitcoins, worth $473 million at the time; they were most likely stolen, probably due to a vulnerability in the protocol (*malleability* will be discussed later on).

Up to 2013, the Silk Road was an online black market, best known as a platform for selling illegal drugs. It was shut down by the FBI in 2013, and the founder, Ross Ulbricht, was sentenced to life in prison.

Even now, ransomware attackers demand payment in Bitcoin, as it is difficult to trace and provides pseudo anonymity through the use of Bitcoin addresses.

7.2 Bitcoin Identity

An easy way to generate new identities in a cryptographic system is to create a new random key-pair, made up of a private —secret— key `sk` and a public key `pk`, which acts as the "public name" of an user. In case `verify(pk, data, sig) == true` then the transaction signed with `sig` was generated by `pk`.

Addresses in the majority of cases represent the owner of a private/public key pair, and are generated using `pk`, but they may also be a **script**.

Anyone can make a new identity at any time, and such identities are not necessarily linked to any real-world identity, but the activity of an identity may be observed over time —the blockchain is in clear readable by anyone— and thus make inferences on to whom it may belong.

[1] Random number

[2] This is a pseudonym, the actual name of the author is unknown

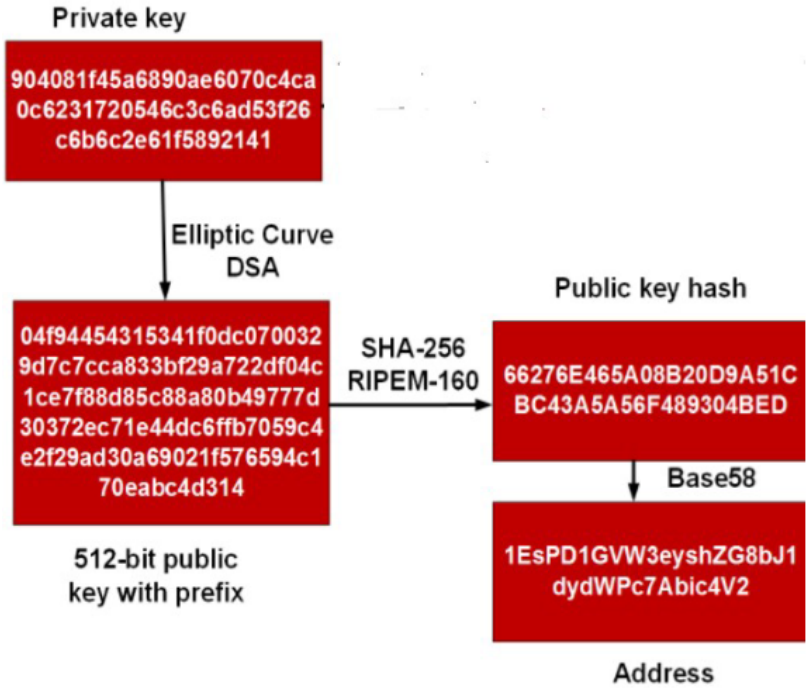


Figure 7.1: Bitcoin address generation process
`Base58` is used to eliminate from the 62 alphanumerical alphabet the characters `(0,O,l,I)` which may appear identical when displayed in certain fonts.

7.3 Bitcoin Transactions

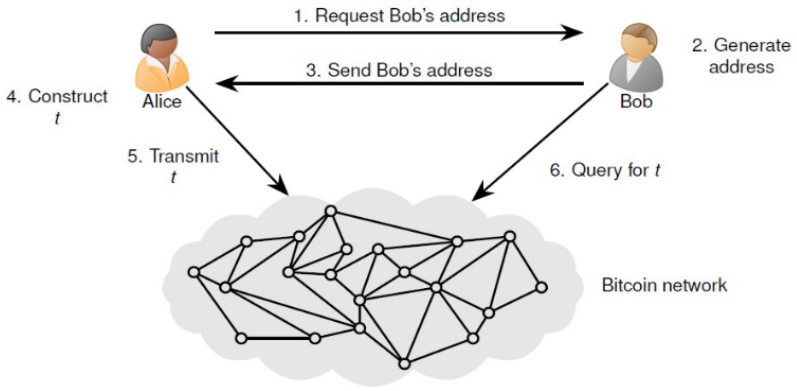


Figure 7.2: Bitcoin transaction workflow
Note that the address exchange is performed out-of-band, not through the Bitcoin P2P network

A transaction t is used to transfer funds from the sender to the receiver, and is registered on the blockchain when confirmed. The transaction t is broadcasted to the network; miners collect broadcasted transactions and include them in a candidate block. When a miner solves the Proof of Work, the block is broadcasted to the network, and —if the block includes t— the transaction t is confirmed.

On the left of each transaction there is a list of inputs, and on the right a list of outputs.

Consider Fig. 7.3: in the left transaction, 0.25 is the money requested by Bob, 1.00 is the money given by Alive and 0.75 is the change she gets back.
Every input must equal the outputs, but note that this should include a **transaction fee**, computed as the difference between the inputs and the outputs, and is given to the miner who confirms the transaction.

It is possible to **merge** or **distribute**, in the first case, multiple inputs are used to create a single output, in the second case, a single input is used to create multiple outputs. So, as said, transactions may also be **multi input** as depicted in Fig. 7.4.

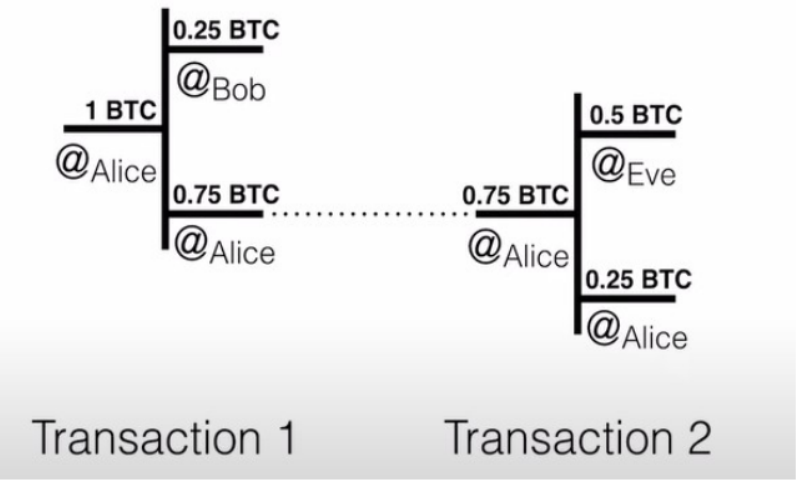


Figure 7.3: Bitcoin linked transactions

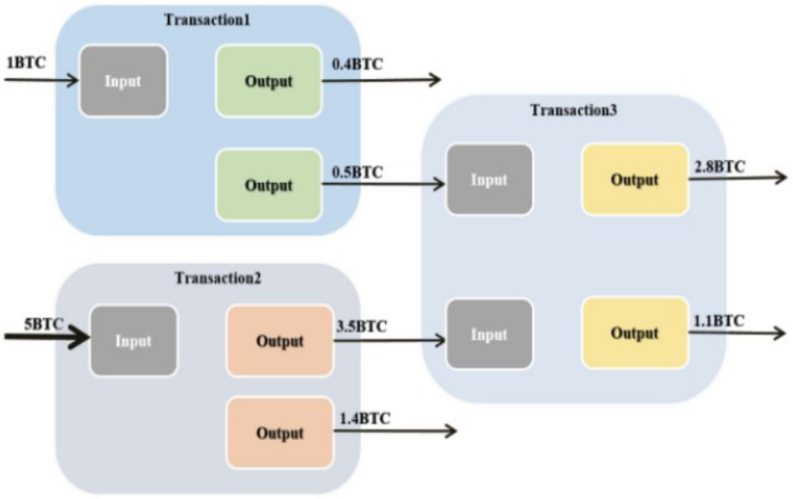


Figure 7.4: Multi input transaction

7.3.1 UTXO Model vs Bank Accounts

Typical centralized currencies exploit "bank" accounts and *transfers* between to move the money from one account ("person") to another. Also Ethereum follows this approach, while Bitcoin adopted the UTXO one, which is based on **unsent output addresses**, i.e. addresses containing bitcoins and not spent in any transaction.

Definition 7.1 (UTXO) *A UTXO represents a certain amount of cryptocurrency that has been authorized by a sender and is available to be spent by a recipient. UTXOs employ public key cryptography to ascertain and transfer ownership. More specifically, the recipient's public key is formatted into the UTXO, thereby limiting the capability to spend the UTXO to the account that can demonstrate ownership of the corresponding private key. A valid digital signature associated with the public key must be included for the UTXO to be spent.*
Wikipedia

Each transaction input is linked (refers) an UTXO of a previous transaction, while each output generates a new UTXO, which is included in the user's wallet and is available to be spent. Each UTXO is *spent* (thus is no more an UTXO) if it is linked to the input of a subsequent transaction.
Getting the the **balance** of an address, equals to scanning the network for UTXOs and adding up all the unspent output locked to that address.

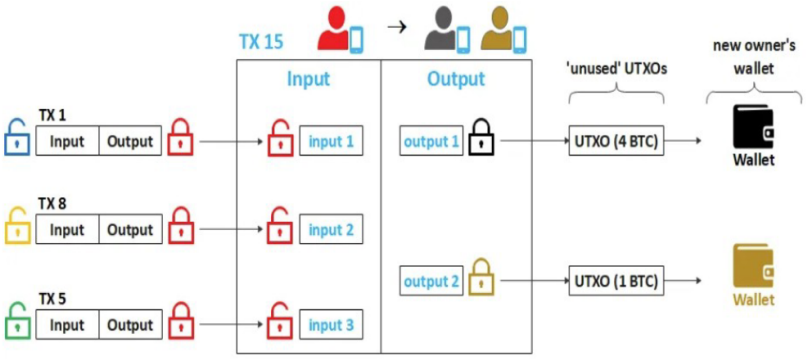


Figure 7.5: UTXO locking

Each UTXO *"locks"* the newly generated UTXO to the new owner's public key, and if they decide to use the UTXO in a new transaction, it must *"unlock"* the funds with their private key.

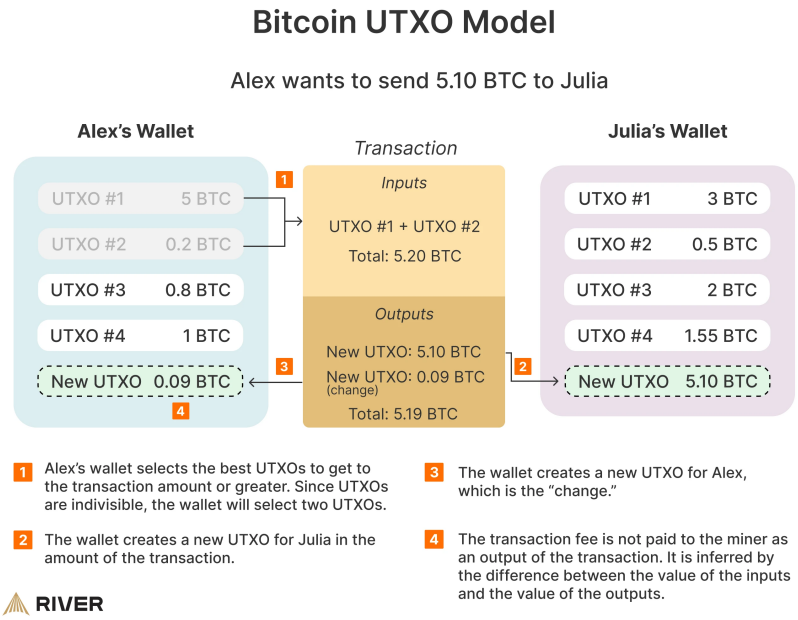


Figure 7.6: More comprehensive example explaining how UTXOs are related to transactions

7.3.2 Scripts

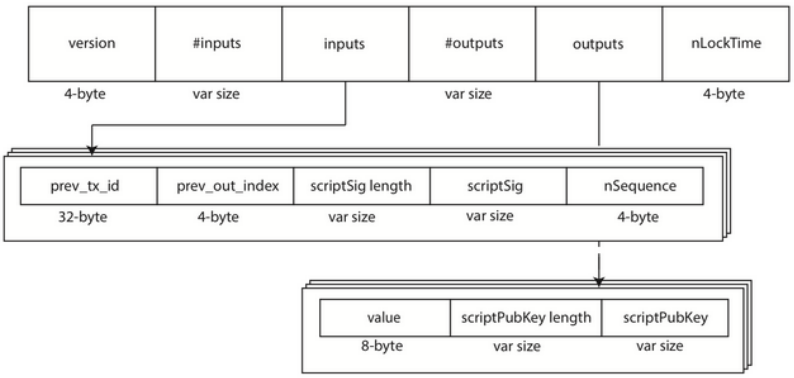
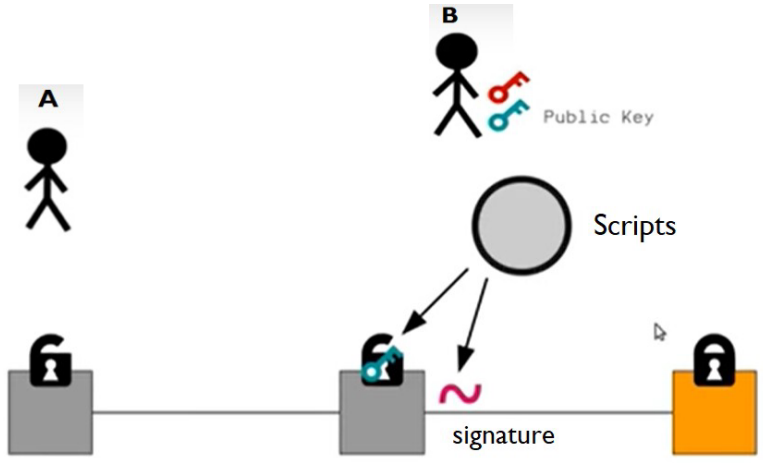


Figure 7.7: Bitcoin Transaction structure

Each transaction contains a **script** written in a simple and —intentionally— limited language[3] to check the ownership of the transferred funds.
Script execution is stateless and completely deterministic. Typical use case is signature verification. Script types include:

- ◇ Pay to Public Key (P2PK)
  Most simple case
- ◇ Pay to Public Key Hash (P2PKH)
  Most common case
- ◇ Pay to Script Hash (P2SH)
- ◇ Pay to Multi-signature

7.3.2.1 Simplified locking and unlocking



- Green: public key (lock)
- Red: signature (unlock)
- locking refers to green, unlocking to red

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.

But what actually happens is this:

[3]Not Turing-complete and overall limited to avoid endless looping and execution errors.

1. *Alice* chooses the UTXO[4] of a transaction t_1 from *Eve* to her. Inside t_1 there is a script that locks the UTXO to *Alice*'s public key.
2. *Alice* creates a new transaction t_2 to send some bitcoin to *Bob*. t_1 is indicated as input for t_2. t_2 includes a script which unlocks the funds from t_1 and locks and a locking script which binds the output of t_2 to *Bob*'s public key.
3. Nodes will verify —among other things— that `eval(t1.locking_script | t2.unlocking_script)` is true.

7.3.2.2 Analyzing P2PKH

As stated before, the outline of a transaction is as follows:
The sender appends a script onto sent bitcoin to lock the transferred bitcoin (the UTXO should include the locking script) and make it usable only by the receiver, which is the only one able to write a script (signature) to unlock the bitcoin. The unlocking script is prepended to the locking script retrieved from the input UTXO, and the result is executed by the Bitcoin network to verify that the sender can actually spend the funds.
The transaction output includes the locking script, which will be used by the recipient when spending the funds. Executing both unlocking and locking scripts allows the receiver to spend the bitcoin.

In the case of P2PKH, the locking script is a `Pay to Public Key Hash` script, which is a script that locks the UTXO to the public key hash of the receiver, i.e. the address. In the default case, the unlocking script is a signature of *all* input and output fields of the transaction, but it's not necessarily like that (See 7.3.2.2).

[4]Or more UTXOs from multiple transaction

Execution is stack based, so first `Sig` and `PubK` are pushed on the stack, then the `OP_DUP` duplicates the top element of the stack, `OP_HASH160` hashes the top element of the stack, `OP_EQUALVERIFY` checks if the top two elements are equal, which are the newly generated hash and the `PubKHash` contained in the locking script, lastly `OP_CHECKSIG` expects `Sig` and `PubK` to be in the stack and verifies the *signature*. The signature may have been performed an all output fields of the transaction, or only to a single one, or none. (See below 7.3.2.2 for more)

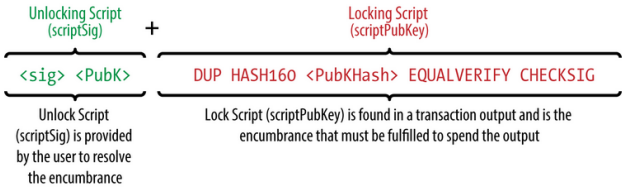


Figure 7.9: Pay to Public Key Hash
Here `PubKHash` is the hash of the public key of the receiver.

Signature types

- `SIGHASH_ALL` signs all the output fields (default)
- `SIGHASH_SINGLE` signs only the output field corresponding to the "current input index"
  *"sign **one** of the outputs– I don't care where the other outputs go."*
- `SIGHASH_NONE` signs none of the output fields
  *"sign **none** of the outputs– I don't care where the bitcoins go"*
- `SIGHASH_ANYONECANPAY` signs only the current *input* (This may be applied after one of the other three)
  "Let other people add inputs to this transaction, I don't care where the rest of the bitcoins come from."

See en.bitcoin.it/wiki/OP_CHECKSIG for more info

To better understand how locking works, check out this stackexchange discussion. It provides a fairly detailed description of the process.
It might be helpful to give a look also at this brief page on the topic.

7.3.2.3 Coinbase Transaction

Coinbase transactions involve fresh bitcoins generated by the system used to reward miners for solving the Proof of Work, and are not linked to any previous transaction.

7.3.3 Transaction Lifecycle

Lifecycle

1. Starts with the transaction's **creation**
2. Then the transaction is **signed** with one or more signatures indicating the authorization to spend the funds referenced by the transaction.
   (See digression on signatures types 7.3.2.2)
3. It is **broadcasted** on the Bitcoin P2P network
4. Each network node (participant) validates and **propagates** the transaction until it reaches (almost) every node in the network.
5. The transaction is verified by a mining node and included in a **block** of transactions recorded on the blockchain.
6. Once recorded on the blockchain and **confirmed** by sufficient subsequent blocks (confirmations), the transaction is a permanent part of the blockchain
7. The funds allocated to a new owner by the transaction can then be **spent** in a new transaction, extending the chain of ownership

Chapter 8

Bitcoin Mining

Recall that the **distributed consesus** is a procedure to reach a common agreement in a distributed or decentralized multi-agent system; it must ensure correct results even in presence of faulty nodes, network partitioning and byzantine faults[1].

Even though it is difficult to classify the method used by Bitcoin to achieve decentralization from a theoretical point of view, it works in practice.

"...is not purely technical, but it's a combination of technical methods and clever incentive engineering."

8.1 Competing

Every node holds a `MemPool` containing all Bitcoin transactions awaiting confirmation. *Conflicts* may happen there, not in the ledger. In case a node receives a double-spending transaction, it will keep the first one and discard the second one.
Nodes try to get transactions into the ledger, and **compete** to do so. Nakamoto consesus is implemented like a "lottery" where the winner gets to add a block —of valid transactions— to the blockchain, and to send its neighbours the updated ledger. The process of competing to add transactions to the blockchain is called **mining**.

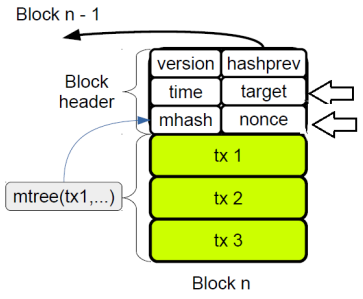
8.1.1 Mining

Mining process starts with filling a candidate block with transactions taken from the memory pool (`MemPool`), and then building a block header (1000 times smaller than the block); finally the node performs the Proof of Work.
So, while a single transaction may be built by anyone, blocks are built by miners, and include multiple transactions and the header.

[1] Nodes behaving maliciously

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



Mining

1. Set `nonce = 0`
2. Hash the *block header* including the `nonce`
3. `while (hash > target)`
   i. Increment `nonce`
   ii. Hash the *block header* including the `nonce`

Target acts as *threshold*, and represents the number of leading zeros the hash must have. The nonce is 32 bit long, and even a slight increment on it changes the whole hash result.

8.1.2 Consesus

Node is selected to propose the next block in proportion to a resource that it is hard to monopolize: in Bitcoin this resource is **computational power** and the selection is done on the basis of the Proof of Work.

The Consensus is **implicit**:
- ◇ No collective distributed algorithm executed by the nodes
- ◇ No voting
- ◇ Selection of malicious nodes is also implicitily handled by the system

Even if the nodes may have occasionally an inconsistent view of the ledger (blockchain forks) consensus will eventually occur, the consistent ledger will eventually be the longest chain.

This is true if the majority of the nodes are honest.

8.1.3 Proof of Work

- ◇ `d` - *difficulty*: a positive number which is used to adjust the time to execute the proof
- ◇ `c` - *challange*: a given string (the block header minus the nonce)
- ◇ `x` - *nonce*: an unknown string

Definition 8.1 (Proof of Work) *A proof of work is a function* $F_d(c, x) \to$ `True,False` *satisfying:*
1. `d` *and* `c` *are fixed*
2. $F_d(c, x)$ *is fast to compute, if* `d`*,* `c`*, and* `x` *are known*
3. *instead, finding* `x` *so that* $F_d(c, x) =$ `True` *is computationally difficult, but feasible.*

The PoW is hard to solve because the computing output looks like a random 256-bit string where each bit is equally likely to be `0` or `1` independently of the other bits, so each output bit looks like coin flips (`0/1`). There is no better way of finding the correct output than trying by **brute force**.
The probability p that the block hash is below the target T and average number of attempts a to find a solution are:

$$p = \frac{T+1}{2^{256}} \qquad a = \frac{1}{p}$$

The system is resistant to Sybil attacks because the PoW is a scarce resource, and the cost of the attack is proportional to the whole computational power of the attacker, not to the number of identities they have.

The Proof of Work is also used in other contexts to prevent spam, like in Hashcash, and to counter DoS attacks, by allowing users to access a service only after solving a PoW.

Email spam may be prevented through a PoW by adding a post stamp to each email message, and the receiver may decide to accept the message only if the PoW is valid.

8.1.4 Block propagation and incentives

8.1.4.1 Block propagation

The mined block is broadcasted on the network, and each node receiving the block verifies that the PoW has been solved by hashing the block header and checking that the hash is less than the target. It is easy to verify, without centralization points.
After the verification, the node adds the block to the blockchain and kicks out any conflicting transaction from `MemPool`.

8.1.4.2 Incentives

There are two mechanisms to incentivize the miners to be honest:

1. **Block reward**: a payment to the miner in exchange for the service of creating a block.
   Bitcoin mints new coins when a new block is mined, and is the only way to create new bitcoins.
   The reward is halved every 210K blocks (~ 4 *years*), and the last block reward will be mined in 2140.
2. **Transaction fees**: for each transaction in the block the miner gets the difference between transaction inputs and outputs.
   It was voluntarily inserted to obtain a good "quality of service" from the miners.

The first transaction in each block is called **coinbase transaction**, and is the one that mints new coins: it includes the reward plus the transaction fees to the miner, and is not linked to any previous UTXO, but to a single "dummy" input.

Mining Difficulty

"To compensate for increasing hardware speed and varying interest in running nodes over time, the proof-of-work difficulty is determined by a moving average targeting an average number of blocks per hour. If they're generated too fast, the difficulty increases."
-Satoshi Nakamoto

The difficulty is adjusted every 2016 blocks (about 2 weeks) to keep the block time around **10 minutes**. The difficulty is adjusted by changing the target, which is inversely proportional to the difficulty.

Why 10 Minutes?

"If broadcasts turn out to be slower in practice than expected, the target time between blocks may have to be increased to avoid wasting resources. We want blocks to usually propagate in much less time than it takes to generate them, otherwise nodes would spend too much time working on obsolete blocks."
-Satoshi Nakamoto

The 10 minutes target time is a trade-off between the time to propagate a block and the time to generate a new one.
The goal was to allow time to propagate across the whole network before the next block gets mined, to avoid wasting resources.

Note that 10 minutes mining time means that no instantaneous transactions are possible, but the system is designed to be secure, not fast.

e.g. you can't pay for an ice cream using bitcoin.

Blocks are preferred over single transactions, because verification is faster and mining single transactions would overall require more mining work.

8.1.5 Tamper-freeness

The blockchain is tamper-free because:

- ◇ The PoW is hard to solve
- ◇ The PoW is easy to verify
- ◇ The PoW is a scarce resource

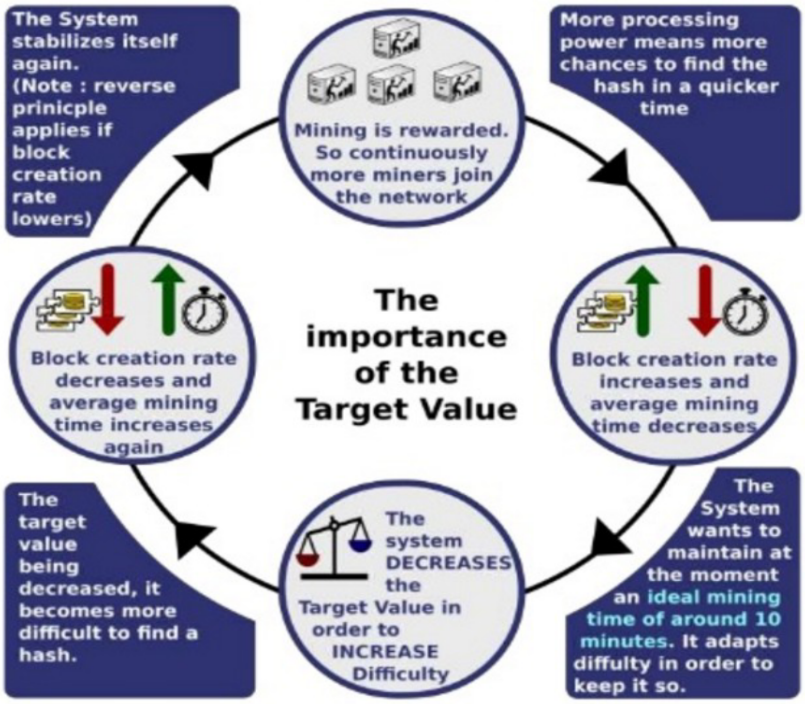


Figure 8.1: Bitcoin target cycle
A node can't avoid decreasing the target, since the other nodes would not validate its Proof of Work.

It would take an unfeasible amount of power for an attacker to change a transaction in a block, since it would imply that:

1. The root of the Merkle tree changes and so the block header
2. The nonce of the block is no more valid
3. Re-execute PoW to re-compute the right nonce for the new block
4. In the next block the hash pointer to the previous block changes as well
5. Nonce of the next block is no more valid
6. Need to re-execute also on next block...
7. ...and so on

8.2 Temporary Forks

Temporary forks may happen when two miners find a valid block at the same time, and broadcast it to the network. The state of the blockchain is seen by the nework consists of two branches both originating from the same parent block. In this case both branches are legitimate, and is different from the double spending case, but still, *which bitcoin are really spent?*

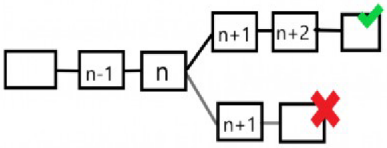


Figure 8.2: Bitcoin temporary forks

Each miner node either receives block A or block C first, which are both valid, but different, and then starts mining on the branch of the fork with the block it received first; note that the two forks may grow indipendently, and the network may have multiple branches. If a miner receives a block that makes the other fork longer, it *abandons* the shorter fork; the transactions of the "abandoned" fork that were not approved in the winning fork are returned to the pool of "not-yet-approved" transactions.

Definition 8.2 (6 Confirmation Rule) *Bitcoin approves a transaction finally only once there are at least* ***five*** *following blocks in the chain*

May the two chains extended perfectly in parallel, to equal height?

It may be possible, but extremely unlikely in practice. The probability that this happens recursively, for a long period is very low, besides mining and the block propagation delay introduce randomness in the protocol that typically prevents this.

Recall that each miner switches to mine on the longest branch it becomes aware of.

Definition 8.3 (Nakamoto Consesus) *Forks are eventually resolved and all nodes eventually agree on which is the longest blockchain. The system therefore guarantees* ***eventual consistency****.*

8.3 Mining technologies

The main Bitcoin actors are:

1. Reference client (Bitcoin Core)
2. Full block chain mode
3. Solo miner
4. Lightweight (SPV) wallet

Basically, the mining process is SHA computation which can be summarized as follows:

```
while (1)
   HDR[kNoncePos]++;
   if (SHA256(SHA256(HDR)) < (65535 << 208)/ DIFFICULTY)
      return;
```

To compute such hash, CPUs were initially used, then GPUs,GPU pools, FPGAs —Verilog programmable hardware— and now ASICs are used. It is unclear what will ASICs successor be. ASICs stands for Application Specific Integrated Circuits, and are hardware devices specifically designed to perform a task, e.g. mining. For this task, they are faster and more efficient than CPUs and GPUs.

8.3.1 Centralized Mining pools

Mining is a very risky task, since it is very likely to spend a lot for mining hardware and eletricity without obtaining a reward for a long time. **Mining pools** are a way to share the risk and the reward among the participants.

The **Pool Manager** sends blocks to miners and distributes the reward among the participants, based on the work they have performed; it must be trusted by anyone.

Challenges

- ◇ How does a pool manager know how much work each member of the pool is actually performing?
  Also miners that have not been able to solve the PoW have to be rewarded for their work.
- ◇ How can the pool manager divide the revenue proportional to the amount of work each miner is doing?
  Partecipants to the pool may cheat, i.e. claim that they've done more than they actually did.
  It is in general hard to prove how much work a node has performed. Generally some "near-valid blocks" are sent to the pool manager, to indicate that the miner is actually working.

Pay-per-share (PPS) is a common reward system used by mining pools, where the pool manager pays a fixed amount for each share submitted by the miner. Miner's are paid from pool's existing balance, and there is no special reward for the miner who actually mines the block.

Pay-proportional is instead a reward system where the pool manager enacts a reward proportional to the amount of work done by the miner, making miners more incentivized to present valid blocks. This is less riskful for the pool manager, since payment is enacted only when a block is mined.

Understading the amount of work done by a miner is still a challenge. *Decentralized mining pools* are a solution to this problem.

8.3.2 Decentralized Mining pools

The basic idea behind decentralized mining pools is to build a separate, private chain including *"weak blocks"* mined with lower difficulty, store in the private chain called *"sharechain"* transactions rewarding for mining weak blocks until a valid block is found; at that point reward is distributed through a side blockchain which is then merged to the main chain, hence obtaining a transparent and fair payout scheme with efficiency minimal performance overhead.

This approach also solves the problem of having a centralization point, since the pool manager does not have to be distribute the rewards.

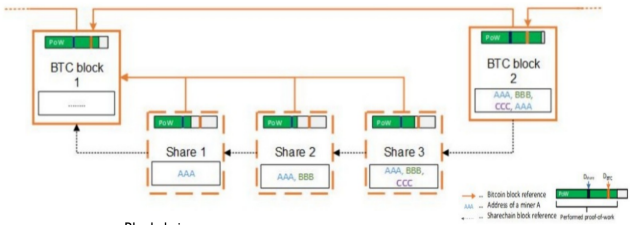


Figure 8.3: Sharechain and main bitcoin chain

Chapter 9

Bitcoin Attacks

Naive attempts to **double spending** are easily detected and rejected by the network. By sending two transactions that spend the same coin, the network will accept only one of them, the first one that arrives. The second transaction will be rejected as it tries to spend a coin that has already been spent. Or, still, adding the same coin two times in the same block will be rejected by the network, as the block would be invalid.
However, there are more sophisticated attacks that can be performed on the Bitcoin network. Later on in this chapter we will discuss some of them.

9.1 Forks

If two blocks A and B are mined at the same time (actually, the second is mined before the first is propagated through the network), the network will have two different versions of the blockchain. This is called a fork.

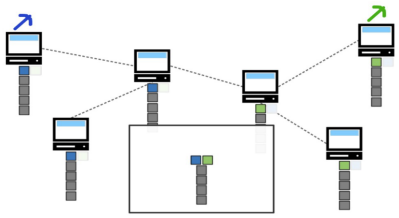


Figure 9.1: Bitcoin Fork

Some nodes will have A as the last block, and some others will have B. The two versions will *coexist* until the next block is mined: if the next block is mined on top of A, then such chain becomes the longest and the network will accept it as the main chain, and the B chain will be discarded, requiring a slight reorganization of the chain. This applies specularly if the next block is mined on top of B.

Forks and double spending

If an attacker performs a double spending attack by sending two transactions that spend the same coin, typically one of the two will be mined before the other (they cannot coexist in the same block), and the latter will result invalid and never mined by nodes. But it may happen that the two transactions are mined simultaneously in two different blocks, resulting in a **fork**. To avoid so, a transaction is accepted by the network only after 6 blocks have been mined on top of it (Def. 8.2).

9.2 51% Attack

This attacks addresses the double spending problem by controlling the majority of the network's mining power. The attacker can then create a "hidden" fork of the blockchain, where the transaction that he wants to double spend is not included. He avoid broadcasting its fork until his chain is longer than the main chain, where he had spent its coins. At that point, he can broadcast his fork —where he still owns its Bitcoins!— and the network will accept it as the longest chain.

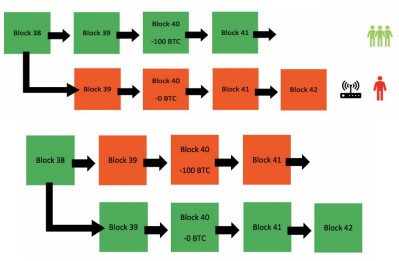


Figure 9.1: Bitcoin 51% Attack

This attack requires for the attacker to control more than 50% of the whole network's mining power, which is a *very difficult* task. It is pretty unlikely that an attacker succeeds.

In 2014 `GHash.io`, a mining pool, reached 51% of the network's mining power. The pool was asked to reduce its power, and it did.

9.3 Transaction Malleability

A subtle attack on the Bitcoin network is the transaction malleability. This attack does not allow to double spend, but it can be used to confuse the network and the users.

Recall that the transaction inputs are signed with the private key of the owners of the funds, allowing to verify the ownership of the funds. All the nodes receiving a transaction verify that the sender(s) are really the owners of the transaction.

A node can change the unlocking script in such a way that the transaction has the same effect and the signature is still valid, but the *TXID changes!* The transaction ID TXID is computed by hashing a set of data that is a superset of the fields covered by the signature. Adding a `NO_OP` operation in the unlocking script, or changing the order of the operations, will change the TXID, but the signature and the unlocking script will still be valid.

9.3.1 Exploiting Malleability

`Alice` issues payment to `Bob` with a transaction $TXID_1$. `Bob` alters the transaction signature before $TXID_1$ is confirmed, and issues a new transaction $TXID_2$. In case $TXID_2$ is confirmed before $TXID_1$, the network will reject the latter, as the funds have already been spent and sent to `Bob`.

`Alice` does not see her transaction confirmed, and she may issue a new transaction to `Bob`, thinking that the first one was not accepted by the network. `Bob` hence receives twice the amount of Bitcoins from `Alice`.

9.4 Segregated Witness

Segregated Witness (`SegWit`) is a soft fork that was activated in August 2017. It was introduced to solve the transaction malleability problem, and to increase the block size limit.

The main idea is to separate the signature data from the transaction data. The signature data is moved to a new structure, the **witness**, that is not included in the transaction hash. This way, transaction ID $TXID$ is not affected by changes in the signature data.

9.5 Lottery is costful

Cheating in the Bitcoin network is very expensive, and most likely leads to a waste of energy and money. But also the honest nodes have to spend a lot of energy to mine a block, and the reward is not guaranteed, making *"solo mining"* not profitable. For this reason, miners join **mining pools**, where they share the reward in proportion to the computational power they provide to the pool.

Recalling Sec. 8.3.1 key points

It is not trivial to demonstrate how much a miner contributed to the pool, and the pool operator could cheat on the miners, by not sharing the reward fairly.

- **Pay per Share** is a method that rewards miners based on the number of shares they contributed to the pool. The reward is enacted immediately, and the pool operator takes the risk of not finding a block.
- **Pay Proportional** rewards miners based on how much they contributed to the pool, but the reward is given only when a block is found. The pool operator takes no risk, but the miners have to trust the operator.
- **Decentralized Mining Pool** a separate private blockchain (`sharechain`, with less difficulty such that valid blocks are mined every 30s) is used to keep track of the shares contributed by the miners, which mine "weak blocks". The pool operator cannot cheat on the miners, and the miners cannot cheat among theirselves, as the sharechain is public and transparent.
  Each weak block refers to a partial resolution of the original PoW, allowing for a **fair** reward distribution. When a miner solves the PoW the last block is linked to the "real" public blockchain, and the reward is shared among the miners.

Chapter 10

Blockchain Scalability

The **Blockchain trilemma** states that blockchain systems can only achieve two of the following three properties: decentralization, security, and scalability. This is because the three properties are in *conflict* with each other, and was initially stated by Vitalik Buterin (Ethereum founder).

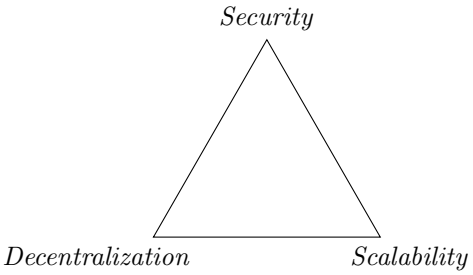


Figure 10.1: Blockchain Trilemma

10.1 "On-chain" Scalability

On-chain scalability refers to the ability of a blockchain to handle a large number of transactions per second. The Bitcoin network can handle 7 transactions per second, while the Ethereum network can handle 15 transactions per second. This is a very low number compared to traditional payment systems like Visa, that can handle 24,000 transactions per second.
Besides note that in Bitcoin 7 transactions per second are handled, but they not get confirmed until 6 blocks have been mined on top of them, taking about 1 hour to actually transfer Bitcoins. Furthermore, the transaction fee is about 55$ to confirm 6 blocks.

On-chain scalability solutions —each one has pros and cons— include:

- Increase block size
- Increase block frequency
- Cross-chains and side-chains
- Use alternative consensus algorithms (e.g. *Proof of Stake*)

10.2 "Off-chain" Scalability

The key idea for **off-chain** scalability is to move transactions off the main blockchain, and to settle them on the main blockchain only when necessary. This way, the main blockchain is not overloaded with transactions, and only used as "arbiter" when necessary.

Off-chain transactions are like promissory notes, where the parties involved in the transaction exchange IOUs, and settle them on the main blockchain only later on. The security of such transactions is almost the same of the main chain, but allow for high-volume of instant, trustless, micropayments.

Potential problems are:

- ◇ Possible **Centralization**
- ◇ **Fund Locking**
- ◇ **Always on requirement**

Bitcoin Lightning Network (Chapter 11) is an example of off-chain scalability solution. It is a second layer protocol that operates on top of the Bitcoin blockchain, and allows for instant, low-fee, micropayments. Currently the Lightning Network can handle 10,000 transactions per second, and it is growing.

10.3 MultiSignature Transactions

`MultiSignature` transactions are a way to increase the security of a transaction, by requiring the signature of multiple parties to spend the funds. This can be used to increase the security of off-chain transactions, by requiring the signature of a third party to settle the transaction on the main chain.

They are implemented through the `CHECKMULTISIG` script, that requires the signature of multiple parties to spend the funds.

Practical `MultiSignature` use cases are:

- ◇ 1-of-n: a transaction requires the signature of only one party to spend the funds.
- ◇ 2-of-2: a transaction requires the signature of both parties to spend the funds.
- ◇ 2-of-3: a transaction requires the signature of two parties to spend the funds.
  Commonly used for *escrow contracts*

10.3.1 Escrow contracts

An **escrow**[1] **contract** is a contract where a third party holds the funds until the conditions of the contract are met. The third party is responsible for releasing the funds to the correct party, and can be used to increase the security of a transaction.

Alice wants to buy a product from Bob, but she does not trust Bob. They can use an escrow contract to secure the transaction, by involving the third —trusted— party Judy:

1. Alice creates a 2-of-3 `MultiSignature` transaction, where the funds are locked in the escrow contract.
2. Coins are held in *escrow* by Alice, Bob and Judy. Any *two* of them can redeem the funds and specify the receiver of the bitcoins

In case there are no disputes, Alice and Bob can redeem the funds together, without involving Judy. If there is a dispute, Judy can decide who "cheated" and who gets the funds.
In either case, two transactions are needed, one from Alice to the escrow, and another one to either Bob or back to Alice.

Technically one among Alice and Bob may decide to give the funds to Judy, but this is not in their interest and would be pointless.

10.4 Towards Pay-to-Script-Hash

`multi-sig` scripts (or transactions?) are longer than standard scripts, require more public keys to be exchanged and come with higher fees. `Pay-to-Script-Hash` (P2SH) is a way to simplify the use of complex scripts, by hashing the script and using the hash as the address. This way, the script is not revealed until the funds are spent, and the address is shorter and easier to use, removing complexity at the sender side. Moreover, the receiver does not need to know the script to redeem the funds.
Furthermore, being the transaction shorter and easier to compute, the **fees** are lower.

10.5 Hash-time locked contracts

Hash-time locked contracts are a way to secure a transaction by locking the funds until a certain time has passed, or a certain condition is met. They are used in the Lightning Network to secure off-chain transactions.

[1] "Escrow" means "deposito in garanzia" in italian

- ◇ `Hash Locks` hash secret and store it in a script. Unlock the funds only if the secret is provided by the intended recipient.
- ◇ `Hash Time Locks` hast secret and store it in a script. Unlock the funds until a certain time has passed, or if the intended recipient provides the secret.

Figure 10.2: Atomic Swap of different cryptocurrencies between Alice and Bob

Bob knows only the secret's hash `19bf...`, not secret itself `XYZ`, so he cannot redeem the funds in Alice's transaction. At this point Bob creates another HTLC using as lock the same hash, and sends it to Alice. Alice can now redeem the funds in Bob's transaction, revealing the secret `XYZ` and redeeming the funds in her transaction. Since the secret is the same and it has been revealed, Bob can now redeem the funds in Alice's transaction.

10.6 Clients against Full Nodes

Full nodes are nodes that store the whole blockchain and validate all the transactions. They are the most secure way to use the Bitcoin network, but they require a lot of resources and time to set up. For the majority of nodes it would be unfeasible to store the whole blockchain which takes up about 500GB of space, as it would be also to use substantial CPU power to validate transactions from other users.

Bitcoin exploits a **Simplified Payment Verification** (SPV) mode, where the client does not store the whole blockchain, but only the block headers (x1000 smaller). This way, the client can verify the transactions without storing the whole blockchain, but it is less secure than a full node.

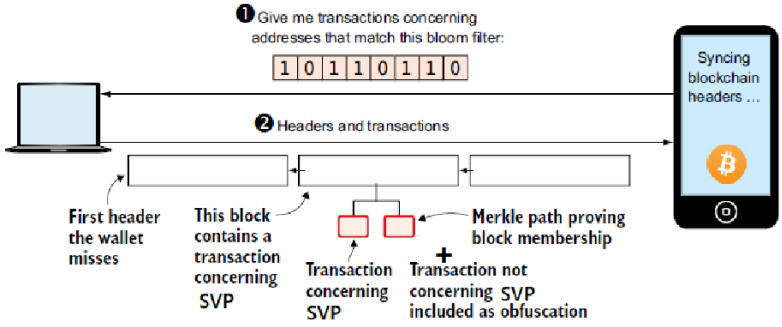


Figure 10.3: SPV client and Full node interaction
The SPV client sends a bloom filter to the full node, that will send back only the transactions that match the filter and Merkle tree paths, along with the block headers not already downloaded by the client.
Actually, the client may be called SVP instead of SPV. What a mess...

The SPV, to check that a given transaction really belongs to a block, computes the Merkle Proof of the transaction exploiting the paths sent by the full node and lastly checks if the Merkle Root is the one expected, which is stored in the block header.

Chapter 11

Bitcoin Lightning network

Bitcoin is currently far from supporting the world finantial exchanges, it handles about 250.000 transactions in a day, while VISA handles about 47.000 transactions *per second*. This is due mainly to max block size and block time, that are respectively 1MB and 10 minutes, and also to the need of broadcasting a transaction to every node. `BitcoinCash` forked from Bitcoin to increase the block size to 8MB, and then to 32MB, but this is not a definitive solution, because the block size can't be increased indefinitely, and the network would be more centralized, since each year every node would have to store about 400TB of data.

11.1 Introduction

The idea behind the **Lightning Network** is to create a second layer on top of the Bitcoin blockchain, where transactions are not broadcasted to the network, but are settled on the main chain only when necessary.

A customer may deposit money on a separate *payment channel*, where payments are registered in private ledgers handled by the "merchant". Every time that the customer buys something, he sends a new transaction replacing the precedent one. The merchant doesn't risk anything because he can commit the transaction on the blockchain at any time, and the customer can't spend more than he has deposited.

In general opening a new channel is a transaction, so it is not optimal to create a new channel every time a customer has to send a payment; this is why the Lightning Network uses a **network of channels**, where a customer can send a payment to a merchant through a series of —existing— channels.

If one of the two parties cheats all the channel funds are granted to counterparty.

11.1.1 How it works

1. *Opening a Lightning Channel*: Let's say Alice wants to pay Bob with Bitcoin. To establish a payment channel, Alice or Bob (or both) must deposit Bitcoin into a 2-of-2 multi-signature (`multisig`) wallet. This is like opening a tab at a bar. You pay some money upfront, and then you can make many transactions (buying drinks) without having to reach for your wallet each time. So Alice sends some money to the multisignature wallet upfront.
2. *Transacting in the Lightning Channel*: Now that there is funding available, Alice can send the payment to Bob. These transactions happen off-chain, meaning they don't need to be recorded on the Bitcoin blockchain. This allows for faster and cheaper transactions. Both of them have a signed copy of the latest state of the channel.
3. *Closing the Lightning Channel*: When Alice and Bob are done transacting, they can close the channel. This involves settling the final state of their transactions on the Bitcoin blockchain. This is like closing your tab at the bar. You settle the final bill based on all the drinks you've had.

In case Bob disappears after Alice has funded the multi-sig address, Alice can get back her funds in 30 days[1].

[1] such delay is introduced to prevent other scams

Example

1. Alice funds the channel with 100 BTC, and Bob with 0 BTC.
2. Alice sends 10 BTC to Bob, and the channel state is updated with a transaction moving from the escrow contract 90 BTC to Alice and 10 BTC to Bob.
3. Alice sends 5 BTC to Bob, and the channel state is updated with a transaction moving from the escrow contract 85 BTC to Alice and 15 BTC to Bob.
4. Bob sends 3 BTC to Alice, and the channel state is updated with a transaction moving from the escrow contract 88 BTC to Alice and 12 BTC to Bob.
5. The channel is closed, and only the last transaction (final state) is settled on the blockchain.

11.2 Punishing cheaters

The first issue is that mid-transactions are valid even in case there are newer ones. There is no way for the Lightning network to guarantee to rip up older transactions when a new one is created. The current workaround is based on **punishment**:

- Transactions have a "revocation secret", which is sent to Bob by Alice when she creates a new transaction.
- In case Alice publishes a newer transaction, Bob can use the revocation secret to redeem all the funds in the channel.

The protocol must give time to Bob to check that Alice is cheating to punish her, so there is usually a 24h delay added to these transactions.

11.3 Multi-hop payments

11.3.1 HTLC

HTLC stands for *Hashed TimeLock Contract*, and is a contract that enforces a transaction to be completed within a certain time frame, or the funds are refunded.

This is used in case multiple channels are involved in a transaction, to ensure that the transaction is completed.

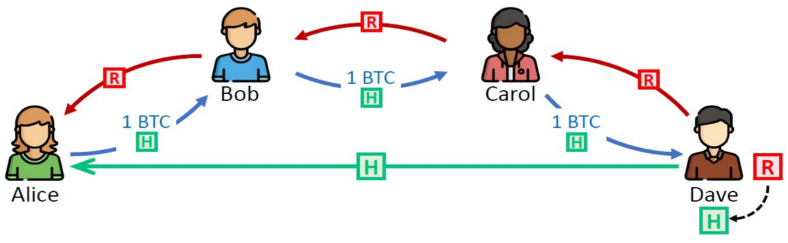


Figure 11.1: Multi hop payments

In this scenario, Alice wants to pay Dave, but they don't have a direct channel, and if for some reason she doesn't want to create a new one, she may exploit existing channels to reach Dave.

To better understand how this works, go back to Fig. 10.2 and read the explanation.

1. Dave generates a secret R and sends $H = hash(R)$ to Alice.
2. Alice sends a *hash-locked* payment to Bob, with the condition that Bob can redeem the funds only if he knows R.
3. Each node generates a *hash-locked* payment to the next node, with the same condition.
4. Dave redeems the payment by revealing R to Carol, who reveals R to Bob, who reveals R to Alice.
5. When Alice knows R, she knows that every node in the path has been paid.

Propagating R

Why does the secret R have to be propagated back? From the moment Dave publishes it, shouldn't it be accessible to all the nodes in the path?

~~I think that the reason may be hidden in the "onion-like" routing schema, which forsees a layer of encryption for each node in the path. In this way, the secret R is not visible to the nodes in the path, and it must be propagated back to the sender. In this way the sender knows that the payment has been completed and that each node in the path has received the BTC, furthermore, anonimity is ensured, every node knows only the previous and the next one.~~

Whether it actually is indeed available to all nodes in the path or not is irrelevant. Dave redeems 1 BTC from Carol by revealing R, and Carol redeems 1 BTC from Bob by exhibiting R, and Bob redeems 1 BTC from Alice by exhibiting R —regardless of whether he had seen it published by Dave, since it's the same secret Carol used to redeem funds from him—. In this way, the payment is propagated back to the sender, so the only one who has actually paid a BTC is Alice.
However due to Onion-Routing described in Sec. 11.3.2, the secret R should not be visible to the nodes in the path, and it must be propagated back to the sender.

The *time-locked* part of the contract is used to ensure that Dave propagates R when he receives its Bitcoin within a certain time frame, if he doesn't the funds are refunded to the transferor.

11.3.2 Routing

The problem of paying someone shifts to **finding a route** between two nodes.
LN uses a **source routing** algorithm, where the source node (each node actually) knows the whole network and can find the shortest path to the destination, it is a BGP-like solution. To ensure privacy, Alice can use **Onion Routing**, where the payment is encrypted multiple times, and each node in the path can only decrypt the outer layer.

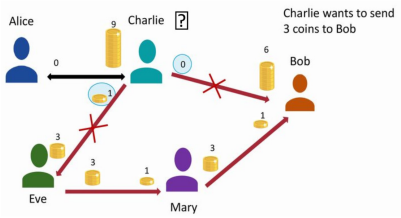


Figure 11.2: Impossible to send `X BTC` from `A` to `B` if `B` holds all the money in the channel

Channels also have **fixed capacity**, established when the channel is opened[2], so the network must be balanced to allow for payments in both directions. If payment fails because of a lack of capacity, the payment is refunded to Alice, and she tries another route.

Finding a route may take minutes.

[2]Depends on how much the channel is funded, and cannot be changed.

11.4 Doubts

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
- If there isn't a watchtower, nodes must be online all the time to protect against fraudolent channel closing; on the other hand the watchtower must be trusted.
- Counterparty misbehaviour can leave coins locked for a long time

Part IV

Ethereum

Chapter 12

Ethereum

Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, fraud or third party interference.
Ethereum is a blockchain platform for building decentralized applications, not only cryptocurrencies, but also:

- ◇ Crowdfunding
- ◇ Tokens
- ◇ Self-sovereign identity (SSI)
- ◇ Supply chains
- ◇ Voting
- ◇ etc.

12.1 Smart Contract

Smart Contract

A **smart contract** is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs.

The smart contract is a piece of code written in `Solidity` automating the "if this happens then do that" part of traditional contracts. It aims to reduce the need for trusted intermediaries.
Ethereum extends to a blockchain supporting distributed data storange and computations.
Compared to Bitcoin's scripting language, Ethereum's language results different on many points:

- ◇ **Turing completeness**: Ethereum's language is Turing complete, meaning that it can solve any computational problem.
- ◇ **State**: Ethereum's language has a state, meaning that it can remember the past.
- ◇ **Blockchain-blindness**: Ethereum's language is aware of the blockchain, meaning that it can read the blockchain's state, e.g. values in block headers.

Smart contracts must be both **transparent** and **flexible**.

Transparency

- ◇ All participants in a blockchain run the same code, each verifying the other
- ◇ Smart contract must be deterministic
- ◇ The logic of the smart contract is visible to all
- ◇ Privacy may be an issue
  solutions based on zero-knowledge proofs may be used in some cases

Flexibility
- ◇ Smart contract are written in a " Turing complete" language
- ◇ Can do anything that a normal computer can do
- ◇ But you need to pay for all nodes on the network to run the code in parallel.
- ◇ Nodes must be rewarded for executing smart contracts
- ◇ Pay for the execution cost

12.2 State machine

Bitcoin's state is held in UTXOs, while Ethereum's state is held in **accounts**, which keep track of balance.
Ethereum has a transaction-based deterministic state machine, and everyone can create its own state transition functions which trigger a state change.

Ether is official cryptocurrency of Ethereum, and is used to pay for transaction fees and computational services on the network.

EOAs (Externally Owned Accounts) are controlled by private keys, and can send transactions to other accounts. They are a bridge from the external world to the internal state of Ethereum.

EVM (*Ethereum Virtual Machine*) is the —"quasi-Turing-complete-machine"— runtime environment for smart contracts in Ethereum. It is completely isolated from the network, filesystem or other processes.
An infinite loop maliciously injected in the EVM would result in a denial of service, so Ethereum has a **gas** mechanism to prevent this. The idea is to pay in *gas* for the contract's execution.

Gas' price is variable, low price means that the transaction will be processed slowly, while high price means that the transaction will be processed quickly; it is up to the sender to decide how much to spend on gas.
The gas **limit** is the maximum amount of gas that the sender is willing to spend on a transaction. If $gas_{limit} \times gas_{price} \geq balance$, the transaction halts.

gas is measured in *gwei*, where $1gwei = 10^{-9}ether$.

12.2.1 Fees and rewards

The transaction fee is calculated as $gas_{price} \times gas_{limit}$, and is paid to the miners, who are more likely to include transactions with higher gas price, as they get more money.

Every operation in the EVM has a fixed gas cost, and if at some point in the execution the gas runs out, the state is reverted, and the sender loses the gas spent, and still pays the fee, which goes to the miners.

12.3 The Merge

On 15th of September 2022, Ethereum Mainnet **merged** with Beacon Chain, resulting in Ethereum 2.0, differentianting from the first version for the consensus mechanism, which is now PoS (Proof of Stake) instead of PoW (Proof of Work).
Proof of Stake is a consensus mechanism that allows the network to reach consensus on the state of the blockchain by allowing the users to vote on the next block, based on the amount of coins they hold, it is different from PoW, where the users have to solve a complex mathematical problem to validate the block.

12.4 Accounts and Transactions

Ethereum accounts have a 20 bytes address and a state, and there may two types:
1. **Externally Owned Accounts (EOAs)**: controlled by private keys, can send transactions to other accounts.
2. **Contract Accounts**: controlled by their contract code, can send transactions to other accounts or create contracts.

12.5 Externally Owned Accounts (EOAs)

EOAs are owned by an external entity, such as a human, and hold:
- ◇ *Address*
- ◇ *Ether balance*
- ◇ `Nonce`, which is the number of transactions sent from the account[1]

[1] Different from the *nonce* in Bitcoin blocks!

- ◇ `storageRoot`: a digest of Ethereum's state, more precisely it is the hash of the root node of a Merkle Patricia tree
- ◇ `codeHash`: the hash of ""(empty string) for EOAs, otherwise hash of the contract account code

The Nonce —somehow— records the order of transactions, and is used to prevent replay attacks and to avoid double-spending; it is incremented every time a transaction is sent from the account.

The *block nonce* is a different thing and it is needed for the PoW (Which actually became a PoS)

Replay attack

Alice signs a transaction —with nonce 22— to send 1 Ether to Bob, and sends it to the network. Bob may intend to *replay* the same transaction to the network and repeatedly submit it to drain Alice's account.

To prevent this, the network checks the nonce of the transaction, and once a transaction with nonce 22 is included in a block, the network will reject any other transaction with nonce 22 coming from Alice. Bob cannot modify the nonce in the replayed transaction, as the signature would be invalid.

EOAs can send transactions to other EOAs to directly send Ether, or may send transaction to trigger a smart contract.

12.5.1 EOA to EOA Transaction

This is the fairly easy case.

1. The sender creates a transaction, signs it with its private key, and sends it to the network.
2. The transaction is broadcasted to the network, and miners include it in a block.
3. The transaction is executed, and the state is updated.

The transaction consists of:

- ◇ Sender's *signature* (`v,r,s`)
- ◇ *Amount*
- ◇ *Receiver* (field `TO`)
- ◇ Other metadata fields, such as *gas price* and *gas limit*

Transactions are serialized using RLP (Recursive Length Prefix).

12.6 Contract Accounts

These accounts contain:

- ◇ *Contract code*
- ◇ Persistent storage of *constant variables*
- ◇ *Ether balance*
- ◇ *Nonce*, which is the number of messages[2] sent from the account
- ◇ They DO NOT have a private key

You need an EOA to create a smart contract, and the contract code is immutable once deployed.
The transaction to do so is the same as the EOA to EOA transaction, but with the `data` field containing the contract code, and with *Receiver* address empty.
Transaction to *interact* with a contract, instead, has the *Receiver* field containing the contract address, and the `data` field containing the method to call and its parameters.

When a contract account receives a message, the code is executed in the EVM, which may perform the following operations:

- ◇ Computation
- ◇ Write to internal storage[3]
- ◇ Send messages to other contracts
- ◇ Create new contracts

[2] Are these transactions?
[3] Internal to who?

Chapter 13

Non Fungible Tokens

This section needs integration and review

Even though the terms "coins" and "tokens" are often used interchangeably, they are not the same. Coins are fungible, meaning that they can be exchanged on a one-to-one basis. Tokens, on the other hand, *may* be non-fungible, meaning that they are unique and cannot be exchanged on a one-to-one basis. This distinction is important because it has implications for how tokens are used and how they are valued.

Digital Tokens are non-native assets created on top of existing blockchains by smart contracts. Fungible tokens are interchangeable (20$ bill may be exchanged for another 20$ bill) and divisible (and mergeable), like cryptocurrencies. A real-world example are casino tips, which are fungible, as they can be exchanged for other tips of the same value.

13.1 Non-fungible tokens

Non-fungible tokens (NFTs) instead are unique and indivisible, like collectibles. NFTs are used to represent ownership of digital or physical assets, such as art, music, real estate, and more. They are created using smart contracts that define the rules for their creation, ownership, and transfer. NFTs are stored on the blockchain and can be bought, sold, and traded like any other asset. They are often used in online games, digital art, and other applications where ownership and scarcity are important.

13.2 ERC standards

The standard was needed to give developers the guarantee that assets will behave in a specific way and to ensure companies that their tokens would be compatible with existing Ethereum infrastructure such as wallets and exchanges.

ERC stands for Ethereum Request for Comments, and it is a set of standards that define how tokens should be created and managed on the Ethereum blockchain. The most popular ERC standards are ERC-20, ERC-721, and ERC-1155. The latter allows for the creation of both fungible and non-fungible tokens, and is used in games and other applications where both types of tokens are needed. Tokens may also act ticket-like, meaning that they are fungible until some event date, and after it they become non fungible collectible tokens.

ICOs (Initial Coin Offerings) are a way to raise funds for a new project by selling tokens to investors. The tokens are usually created using the ERC-20 standard, which allows them to be traded on cryptocurrency exchanges. The ERC-20 standard defines how tokens should be created, transferred, and managed on the Ethereum blockchain. It also includes rules for how tokens should be stored and how they should be transferred between accounts.

DeFi (Decentralized Finance) tokens are used as "collateral" in lending and borrowing platforms, where users can borrow funds by depositing their fungible tokens as "collateral". They are used also for voting and governance in decentralized organizations, but also as a simple medium of exchange in financial transactions.
They represent is a new way of doing finance that uses blockchain technology to create decentralized financial products and services. DeFi is built on the Ethereum blockchain and uses smart contracts to automate financial transactions. DeFi is often used to create lending and borrowing platforms, decentralized exchanges, and other financial products that are not controlled by a central authority.

13.2.1 ERC-20

ERC-20 is the most widely used standard for creating tokens on the Ethereum blockchain. It defines a set of rules that tokens must follow in order to be compatible with the Ethereum ecosystem. These rules include how tokens should be created, transferred, and managed, as well as how they should be stored and how they should be transferred between accounts. ERC-20 tokens are **fungible**, meaning that they can be exchanged on a one-to-one basis. They are often used in ICOs and other applications where fungibility is important.

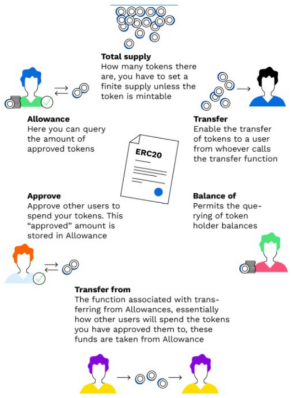


Figure 13.1: ERC20 recap

The token **contract** contains a map of account addresses and respective *balances*. The balance may differ from contract to contract, it may represent an amount of physical objects, rights, monetary value, etc.

ERC-20 tokens must implement the following functions:

- `totalSupply()`: returns the total supply of tokens
- `balanceOf(address owner)`: returns the balance of tokens for a given address
- `transfer(address to, uint256 value)`: transfers tokens from the sender to the given address
- `transferFrom(address from, address to, uint256 value)`: transfers tokens from one address to another
- `approve(address spender, uint256 value)`: approves the given address to spend the sender's tokens
- `allowance(address owner, address spender)`: returns the amount of tokens that the spender is allowed to spend on behalf of the owner

Optional features include the `name()`, `symbol()`, and `decimals()` functions, which return the name, symbol, and number of decimal places for the token, respectively.

Allowances are used to allow one address to spend tokens on behalf of another address. This is useful for applications where tokens need to be transferred between accounts without the sender having to approve each transfer individually. They are mostly used in decentralized exchanges (`DeFi` applications), where users can trade tokens without having to trust the exchange with their funds.

Allowance example

- consider user A (Alice) and user B (Bob).
- A has 1000 tokens and want to give permission to B to spend 100 of them.
- A calls approve(address(B), 100)
- B checks how many tokens A gave him permission to use by calling allowance(address(A), address(B))
- B sends to his account some of these tokens by calling transferFrom(address(A), address(B), 50)
- in subsequent withdrawals, B can finish withdrawing the rest of the funds, but he can only go up to 100 tokens, in total

Minting and burning are the processes of creating and destroying tokens, respectively. These processes are often used in ICOs and other applications where tokens need to be created or destroyed on demand. The `mint()` and `burn()` Solidity (or OpenZeppelin) functions are used to create and destroy tokens, respectively.

13.2.2 ERC-721

ERC-721 is a standard proposed in 2018 for creating non-fungible tokens on the Ethereum blockchain. It defines a set of rules that tokens must follow in order to be compatible with the Ethereum ecosystem. These rules include how tokens should be created, transferred, and managed, as well as how they should be stored and how they should be transferred between accounts. ERC-721 tokens are **non-fungible**, meaning that they are unique and cannot be exchanged on a one-to-one basis. They are often used in online games, digital art, and other applications where ownership and scarcity are important.

All NFTs have a unique identifier `uint256` called `tokenId`, which is used to distinguish them from other tokens. This identifier is often a hash of the token's metadata, which includes information about the token's creator, owner, and other attributes. The metadata is stored off-chain and can be accessed by anyone who has the token's unique identifier.
For each NFT the pair `(address, uint256)` must be globally unique, with the `address` being the owner of the token.

Transferring NFTs

- The owner —Bob— of an NFT can transfer it to another address by calling the `transferFrom(Bob,Alice, ID)` function on the token's contract. The ricipient address may be either an EOA or a contract, but in the latter case such contract must implement the `onERC721Received()` function, which provides an acknowledgement instrumental to the transaction's success.
- The owner of an NFT can also give permission to another address to transfer the token on their behalf by calling the `approve()` function on the token's contract.
  The approved address can then transfer the token to another address by calling the `transferFrom()` function.

Ongoing royalties can be implemented by the contract, which will automatically send a percentage of the sale to the original creator of the NFT. This is done by implementing the `royaltyInfo()` function, which returns the percentage of the sale that should be sent to the creator.
On openSea the royalties are automatically managed by the platform on NFTs creation, and the creator can set the percentage of the sale that they want to receive as royalties.

The contract may include —unlockable— content that only the NFT owner can access, such as a link to a digital file or a message from the NFT creator. This is done by implementing the `tokenURI()` function, which returns the URI of the token's metadata. The metadata is stored off-chain and can be accessed by anyone who has the token's unique identifier.

Contract metadata is a JSON object that contains information about the token, such as its name, symbol, and other attributes. The metadata is stored off-chain and can be accessed by anyone who has the token's unique identifier. The metadata is often stored on IPFS or another decentralized storage platform to ensure that it is always available.

Storing the data on-chain would be too expensive, as it would require a separate transaction for each token. It is used only if there is some information which must be persistently stored on-chain, such as the token's creator or owner.

Chapter 14

Solidity Attacks

Solidity is a high-level language whose syntax is similar to that of JavaScript. It is designed to target the Ethereum Virtual Machine (EVM). Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features. It is used to implement smart contracts on various blockchain platforms, the most popular of which is Ethereum.

14.1 DAO attack

The DAO (Decentralized Autonomous Organization) was a smart contract that was created to act as a venture capital fund for the Ethereum ecosystem. It was launched in April 2016 and raised over $150 million in Ether, making it the largest crowdfunding campaign in history at the time. The DAO was designed to allow investors to vote on which projects to fund and to receive a share of the profits from those projects.

The DAO was implemented as a smart contract on the Ethereum blockchain using Solidity. The contract was designed to allow investors to deposit Ether into the contract in exchange for DAO tokens, which could be used to vote on proposals for funding. The contract also contained a function that allowed investors to withdraw their Ether at any time.

In June 2016, an attacker exploited a vulnerability in the DAO contract that allowed them to drain funds from the contract. The attacker used a recursive call attack to repeatedly call the withdraw function on the contract, draining over $50 million in Ether before the attack was stopped.

14.1.1 Reentrancy attack

The **reentrancy attack** is a type of attack that exploits a vulnerability in a smart contract that allows an attacker to repeatedly call a function on the contract before the previous call has completed. This can allow the attacker to drain funds from the contract or perform other malicious actions.

The DAO attack was a reentrancy attack that exploited a vulnerability in the DAO contract that allowed the attacker to repeatedly call the withdraw function on the contract before the previous call had completed. This allowed the attacker to drain funds from the contract by repeatedly withdrawing Ether from the contract.

To prevent it from happening, the contract should be designed in such a way that it is not possible for an external contract to call the withdraw function before the previous call has completed. This can be done by using the `require` statement to check that the contract is in a valid state before allowing the function to proceed, or using `send()` or `transfer()` instead of `call.value()`.

14.2 Arithmetic overflow and underflow

Arithmetic overflow and underflow are common vulnerabilities in smart contracts that can lead to unexpected behavior and security issues. An overflow occurs when the result of an arithmetic operation exceeds the maximum value that can be stored in the data type used to represent the result. An underflow occurs when the result of an arithmetic operation is less than the minimum value that can be stored in the data type used to represent the result.

A good prevention strategy is to use the SafeMath library, which provides functions for performing arithmetic opera-

tions that check for overflows and underflows and revert the transaction if an overflow or underflow is detected. Never use Solidity arithmetic operators directly, always use SafeMath functions.

14.3 Phishing

Phishing is a type of attack where an attacker tries to trick a user into revealing sensitive information, such as passwords or private keys, by impersonating a legitimate entity. Phishing attacks are common in the cryptocurrency space, where attackers try to steal funds by tricking users into revealing their private keys or other sensitive information.

Chapter 15

Blockchain Applications

Aside from cryptocurrencies, blockchain technology has many other applications. In this chapter, we will explore some of the most popular blockchain applications and how they are being used today.

15.1 Supply Chain Management

Blockchain can provide increased supply chain transparency, as well as reduced cost and risk across the supply chain. This is because blockchain can provide a single source of truth for all parties involved in a supply chain, which can help to reduce disputes and errors. In addition, blockchain can help to improve traceability and accountability in the supply chain, which can help to reduce fraud and counterfeiting.

A centralized database maintained by a manufacturer designed to track components, equipments, locations, service histories, etc. may result in data silos, data duplication, data inconsistency, and may potentially be a single point of failure: data tampering may result in serious business damage for the manufacturer. A blockchain-based system instead is both **distributed** and **tamper-proof** can provide a single source of truth for all parties involved in a supply chain, which can help to reduce disputes and errors.

15.2 Voting Systems

Missing section

15.3 NFTs

NFTs (Non-Fungible Tokens) are unique digital assets that are stored on a blockchain. They can represent anything from digital art to virtual real estate to collectibles. NFTs are created using smart contracts, which are self-executing contracts with the terms of the agreement between buyer and seller being directly written into lines of code. This makes NFTs secure, transparent, and tamper-proof.

Common use cases are:

- **Gaming**
  Tradeable in-game assets, such as weapons, armor, and skins, are popular NFTs in the gaming industry. Players can buy, sell, and trade these assets on blockchain-based marketplaces.
- **Collectibles**
  An example are baseball cards, becoming popular in the last years.
- **Art**
  Rarible and *OpenSea* are popular marketplaces for NFT art. NFTs allow to monetize digital art in a way that was not possible before.
- **Virtual Assets**
  Domains such as `.eth` and `.crypto` have been turned into NFTs.
  Decentraland and *CryptoVoxels* are examples of virtual worlds where users can buy, sell, and trade virtual real estate.
- **Real-world Assets**
  OpenLaw is a platform that allows users to create and manage legal agreements using blockchain technology.

Users can create NFTs that represent real-world assets, such as real estate, cars, and intellectual property. *Nike* has patented a system to tokenize shoes.

- ◇ **Identity**
  NFTs would allow to store identity information on the blockchain, such as passports, driver's licenses, birth certificates, and medical history.

15.4 Buying and Selling NFTs

To buy and sell NFTs, you need to use a blockchain wallet that supports NFTs, such as MetaMask. You can connect your wallet to a blockchain-based marketplace, such as OpenSea or Rarible, and browse the available NFTs. When you find an NFT that you want to buy, you can place a bid or purchase it outright using cryptocurrency. Once you own an NFT, you can sell it on the marketplace or transfer it to another wallet.
There is no need to know any contracts programming knownledge to mint[1], buy and sell NFTs.

15.5 Identity Management

SSI (Self-Sovereign Identity) is a concept that allows individuals to control their own digital identities. This is done by using blockchain technology to create decentralized identifiers (DIDs) and verifiable credentials. DIDs are unique identifiers that are stored on a blockchain and can be used to prove ownership of a digital identity. Verifiable credentials are digital certificates that can be used to prove claims about an individual, such as their age, address, or qualifications. SSI allows individuals to share their digital identities with others in a secure and privacy-preserving way.

A DID document is a JSON object that contains information about a DID, such as its public key, authentication methods, and service endpoints. Verifiable credentials are JSON objects that contain claims about an individual, such as their name, date of birth, or address. Verifiable credentials are signed by the issuer and can be verified by the recipient using the issuer's public key.
A DID (*Decentralized Identifier*, i.e. key?) resolves to an abovementioned DID document, which, more precisely, contains:

1. DID
2. Set of public keys
3. Set of authentication methods
4. Set of service endpoints
5. Timestamp of the document
6. Signature of the document

[1] i.e. Create

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 cryptographic puzzle, and the first one to solve it gets to add a new block to the blockchain. This process is energy-intensive and requires a lot of computational power, and has its cost has risen so much that mining pools (*farms*) control a significant portion of the network's hash rate, ultimately making the blockchain not fully distributed.

It is important to understand that the Proof of Work algorithm is not the only way to reach consensus in a blockchain network.
Proof of Stake is an alternative consensus algorithm that is more energy-efficient and less centralized than Proof of Work. In a Proof of Stake system, validators are chosen to create new blocks based on the number of coins they hold. This means that the more coins a validator holds, the more likely they are to be chosen to create a new block.

There are also Delegate Proof of Stake (DPoS, used by Algorand, Cardano, ...) and byzantine consensus (Hyperledger) algorithms.

Definition 16.1 (Ethereum PoS) *The Proof-of-Stake (PoS) Ethereum consensus protocol is constructed by applying the finality gadget Casper FFG on top of the fork choice rule LMD GHOST, a flavour of the Greedy Heaviest-Observed Sub-Tree (GHOST) rule which considers only each participant's most recent vote (Latest Message Driven, LMD).*

This does not have to be known. It is just a definition to introduce *GHOST* and *Casper* acronyms.

16.1 Validators opposed to Miners

PoS introduces the role of *Validators* opposed to *miners*; they propose new blocks, but without solving the PoW, and cast a vote to decide which will be the next block of the blockchain.
Note that a single node can host anything from zero to hundreds or thousands of validators, and they would all share the same local view of the blockchain.

16.1.1 Being a validator

The Key point here is not solving a puzzle, but to use the cryptocurrency itself as a stake to validate transactions; validators have put something of value into the network that can be destroyed if they act dishonestly. To participate as a validator, a user must deposit 32 ETH into the deposit contract and run three separate pieces of software: an *execution* client, a *consensus* client, and a *validator* client.

Validators are chosen to be in a **committee**, and they are responsible for proposing and voting on new blocks. The committee is chosen randomly, and the size of the committee is proportional to the total amount of ETH staked in the network. One peer among them will be chosen as the *proposer*, and the rest will be *validators*. For a block to be inserted in the blockchain, it must be signed by 2/3 of the validators in the committee (super-majority).

PoS, unlike PoW, is related to time and has to order events in **epochs**. For each epoch there are 32 block proposers, and each proposer is responsible for proposing a block in a slot.

Each validator must assess the validity of the block proposed by the proposer, and if it is valid, they sign it, enacting an **attestation**. Each validator can attest one block per epoch, and if they do not, they get *slashed*.

Can attest or must attest? I think it is the latter, due to the definition of slashing below.

16.2 Nothing at Stake

The *Nothing at Stake* problem is a theoretical problem that arises in PoS systems. It is based on the idea that validators have nothing to lose by voting for multiple blocks in a blockchain fork, as opposed to PoW, where miners have to choose one chain to mine on.

To overcome this problem, Ethereum uses a mechanism called *slashing*, which penalizes validators who vote for multiple blocks in a fork. Validators who are caught voting for multiple blocks are penalized by having a portion of their stake slashed. The same applies for proposing multiple blocks in the same slot.

Slashing is possible because validators apply a signature to the block they propose, so if multiple blocks appear to be signed by the same malicious validator, he is caught and penalized.

16.3 *Gasper* - Casper and GHOST

PoS consesus algorithm in Ethereum is called **Gasper** which results from putting together two main components: Casper and GHOST.

1. **Casper** - checkpoint candidate choice: defines whether a previously accepted block will become the next global checkpoint.
   Casper is a finality gadget that ensures that once a block is added to the blockchain, it cannot be reverted. This is done by having validators vote on the validity of blocks, and once 2/3 of the validators have voted on a block, it is considered final.
2. **GHOST** - fork choice rule: a new rule to choose the branch in case of fork, different from the longest chain rule

16.3.1 GHOST

GHOST is a fork choice rule that considers only each participant's most recent vote (Latest Message Driven, LMD). It is a rule that considers the weight of the blocks in the blockchain, and not just the length of the chain. This means that a block that has more weight (i.e., more validators have voted for it/block has more attestations) is considered to be the main chain, even if it is shorter than another chain.

The winning branch is the branch with the most accumulated staked ETH that has voted for it.

16.3.2 Finality

Finality is the property that ensures that once a block is added to the blockchain, it cannot be reverted. This is done by having validators vote on the validity of blocks, and once 2/3 of the validators have voted on a block, it is considered final.

In Bitcoin there is no finality, and a block can be reverted if a longer chain is found. In Ethereum, once a block is added to the blockchain, it is considered final and cannot be reverted.

Periodically, all honest validators agree on fairly recent checkpoint blocks that they will never revert, i.e. no forks are possible before a checkpoint

Chapter 17

ZeroKnowledge Proofs

In ZeroKnowledge Proofs (ZKPs), a prover can convince a verifier that a statement is true without revealing any information about the statement itself. This is done by proving that the prover knows a secret without revealing the secret itself.

17.1 Introduction

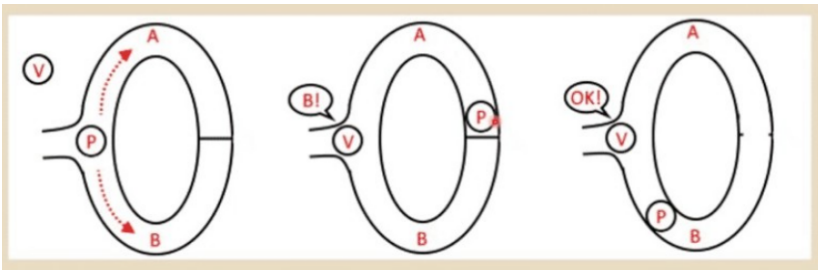


Figure 17.1: Here V acknowledges that P knows the secret because he sees P coming out of B

The "Alibaba's cave" is a common example to explain ZKPs. In this scenario, a cave has a door that can only be opened by a secret password. A prover wants to convince a verifier that they know the password without revealing it. The prover can open the door and show the verifier that they know the password, but the verifier cannot see the password itself.
But what if the prover simply had luck and the door has opened by itself? To avoid this, the prover can repeat the process multiple times, proving to the verifier that they know the password with a high probability.

17.1.1 Properties of ZKPs

- **Completeness**: if the statement is true, the verifier will be convinced by the prover.
- **Soundness**: if the statement is false, the prover will not be able to convince —easily— the verifier.
  its luck will eventually run out
- **Zero-Knowledge**: the verifier learns nothing about the statement itself.

17.1.2 Real World and Blockchains

An interactive protocol is not so useful in the real world, as it often limits the number of potential use-cases of the primitive: RTTs are expensive and slow, and the prover and verifier would have to be online at the same time.

For what concerns privacy-protected transfers in Blockchains, using ZKPs, it is possible to prove that someone is allowed to spend a certain amount without revealing the sender and receiver address and amount to the whole network.

17.2 ZK-SNARKs

Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (ZK-SNARKs) are a type of ZKP that allows a prover to convince a verifier that a statement is true without revealing any information about the statement itself.

ZK-SNARKs are **non-interactive**, meaning that the prover can generate a proof and send it to the verifier without any interaction.

- ◇ `ZK` *Zero-Knowledge*: the verifier learns nothing about the statement itself
- ◇ `S` *Succinct*: the proof is short and easy to verify
- ◇ `N` *Non-Interactive*: the prover can generate a proof and send it to the verifier without any interaction
- ◇ `ARKS` *Argument of Knowledge*: the prover convinces the verifier that the statement is true

17.3 XACML and ZKPs

XACML (eXtensible Access Control Markup Language) is a standard for access control that allows administrators to define policies that determine who can access what resources. ZKPs can be used to prove that a user is allowed to access a resource without revealing the user's identity or the resource being accessed.

Prof. Ricci et al. proposed a XACML framework exploiting a blockchain to implement access control, removing the need for a trusted third party to perform ACAB (*Attribute Based Access Control*) and using smart contracts instead

- ◇ Outsource the access control decision process
- ◇ No need of a trusted third party to perform the access control decision process
- ◇ Auditability, decentralization

The initial proposal however posed the problem of privacy, as the blockchain is public and all the transactions are visible to everyone. To solve this, ZKPs can be used to prove that a user is allowed to access a resource without revealing the user's identity or the resource being accessed, allowing for the presence of "private attributes". The second solution exploited Zokrates, a toolbox for zkSNARKs on Ethereum, to implement the ZKPs.