Peer to Peer - Appunti

Francesco Lorenzoni

February 2024

Contents

Part I

Introduction to P2P Systems

Course info

...

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*

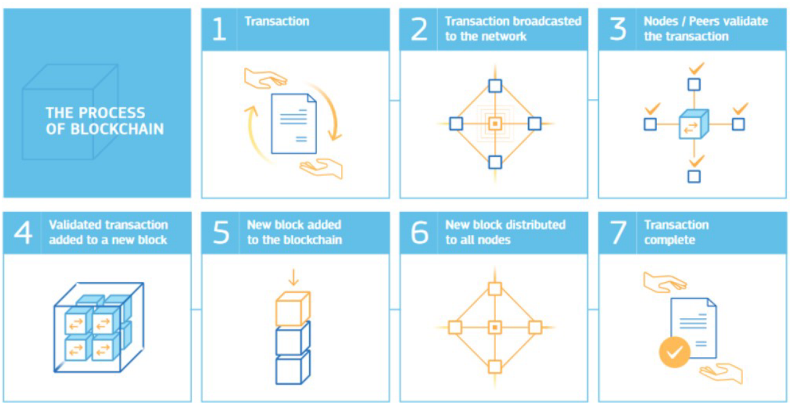


Figure 1.1: Blockchain process

[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 server 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.

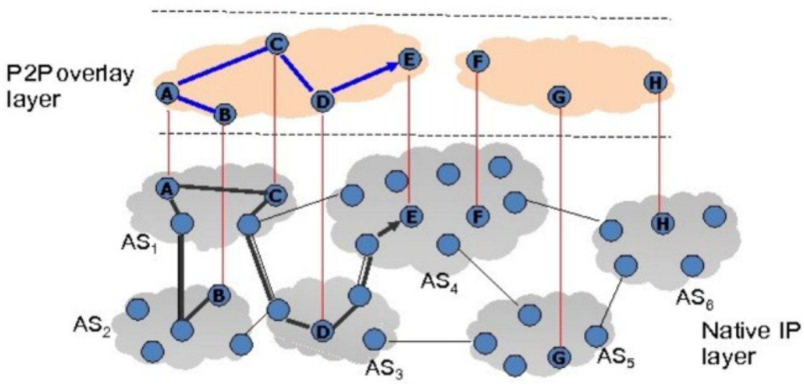


Figure 1.2: P2P Overlay networks

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

1.3.1 Unstructured overlay

[2]Non Fungible Tokens

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`, `BitTorrent`, `BitCoin`

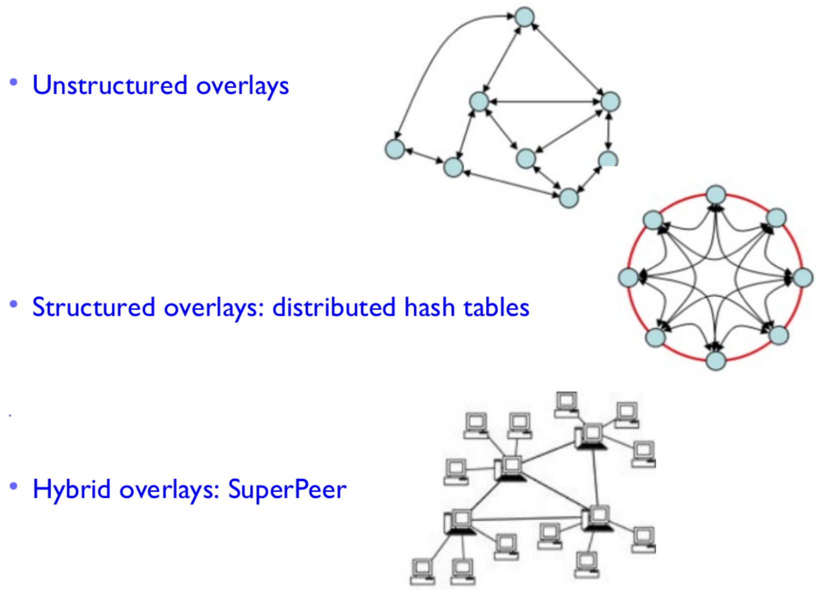


Figure 1.3: P2P Overlay Network classification

Flooding

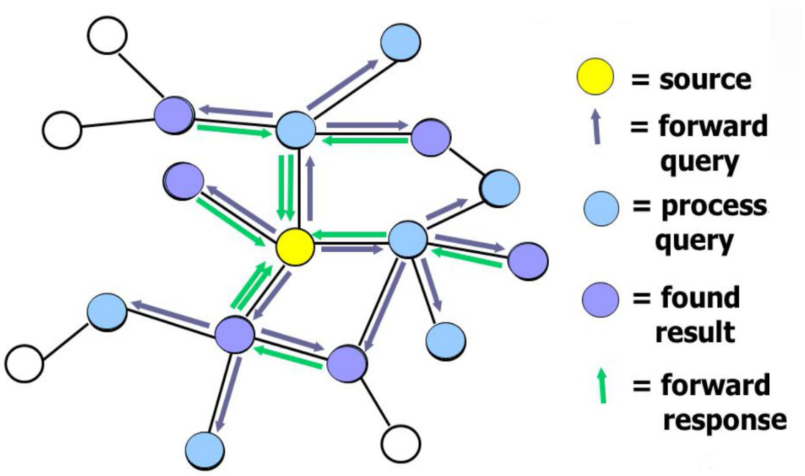


Figure 1.4: Flooding search in unstructured overlay

Messages have a *TTL* to limit the number of hops when propagating, but also a *unique identifier* to detect cycles.

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

Expanding Ring

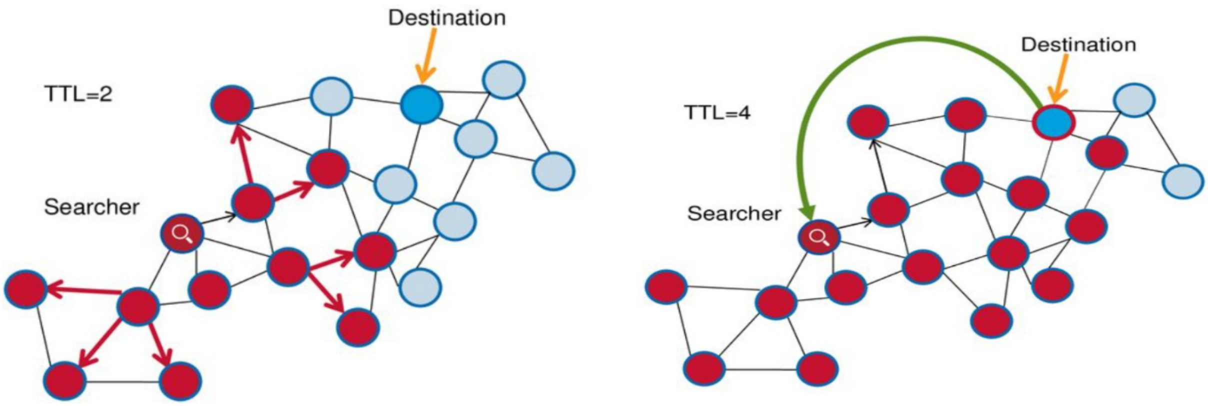


Figure 1.5: Expanding ring/Iterative Deepening
This tecnique consists in repeated flooding with an increasing TTL, implementing a BFS search.

Random walk

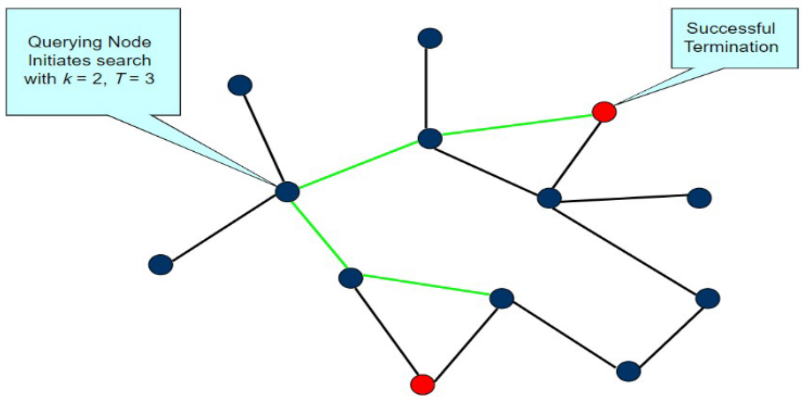


Figure 1.6: Random Walk
k indicates the number of "walkers" to be generated by the querying node. Each green path corresponds to a "walker".
A path is constructed by taking successive (single) steps in random directions defined by a *Markov Chain*, which is "memory-less". Note that only one successor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

Random walk avoids and exponential increase in the amount of messages, 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

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. $\mathcal{O}(logN)$

1.3.3 Hierarchical overlays

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.

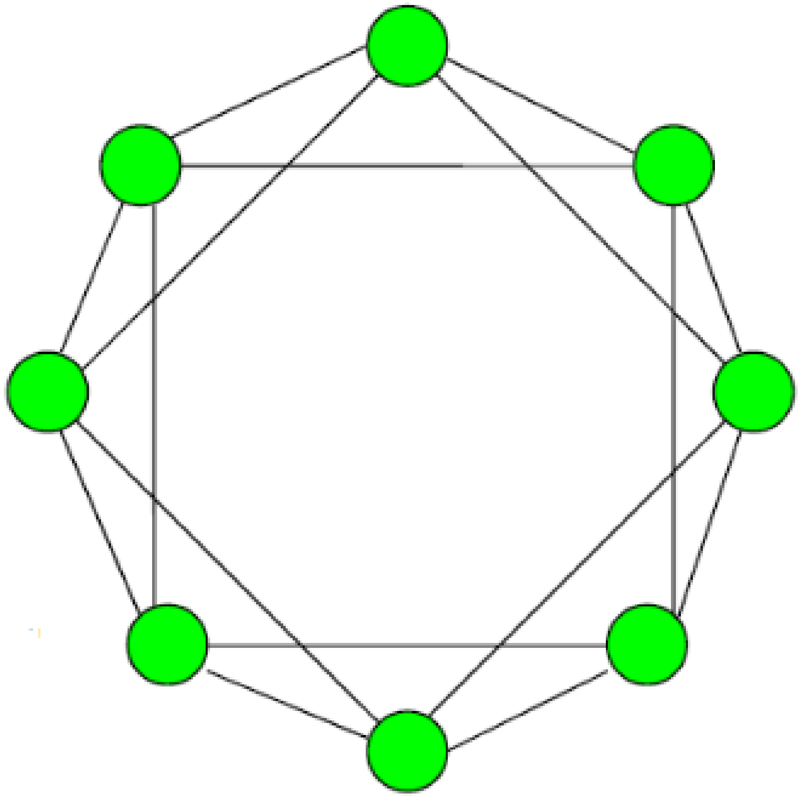


Figure 1.7: Structured overleay

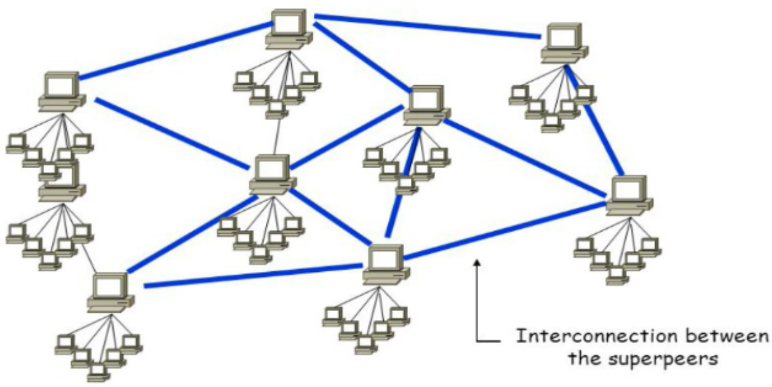


Figure 1.8: Hierarchical overlay

1.3.4 Summary

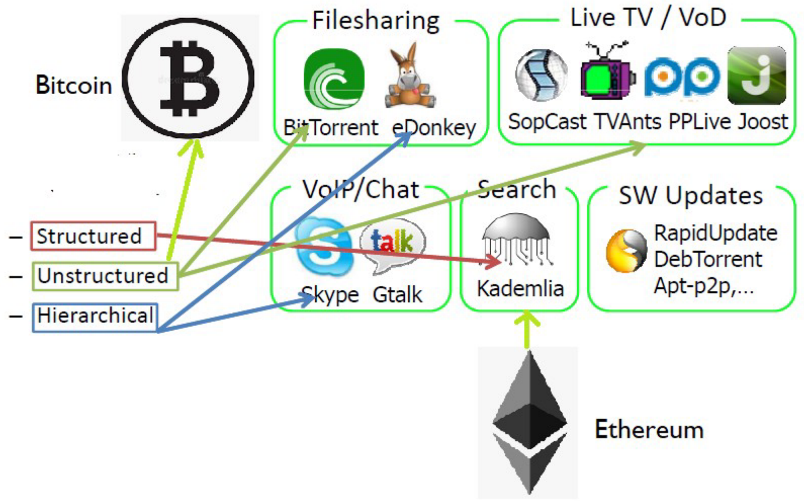


Figure 1.9: Overlay structure for known applications

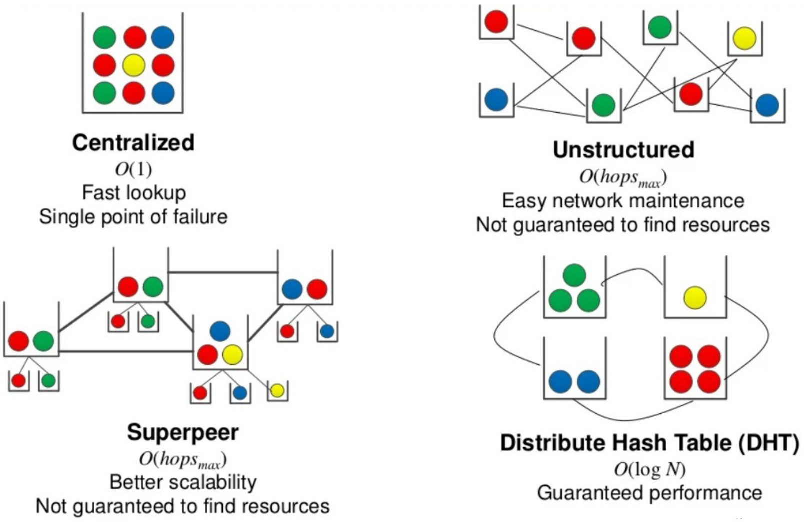


Figure 1.10: Overlays summary
DHT will be discussed in the next chapter

Chapter 2

Distributed Hash Tables

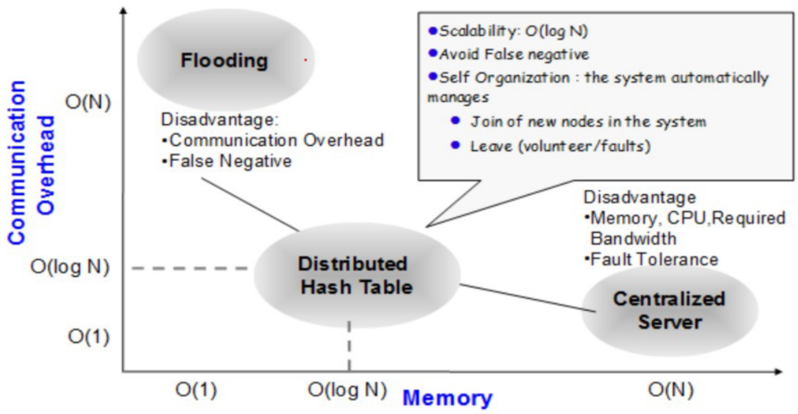


Figure 2.1: DHT Motivations

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.

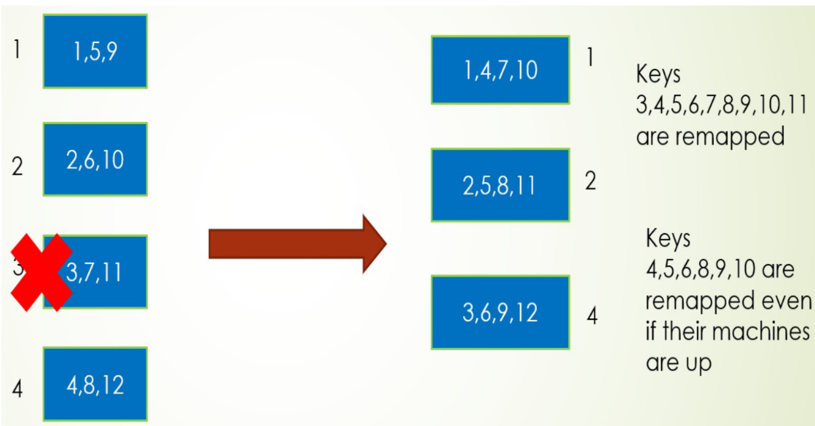


Figure 2.2: Rehashing problem

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 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 H.

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

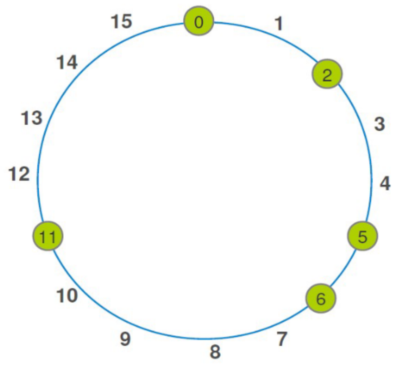


Figure 2.3: Identifier space

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

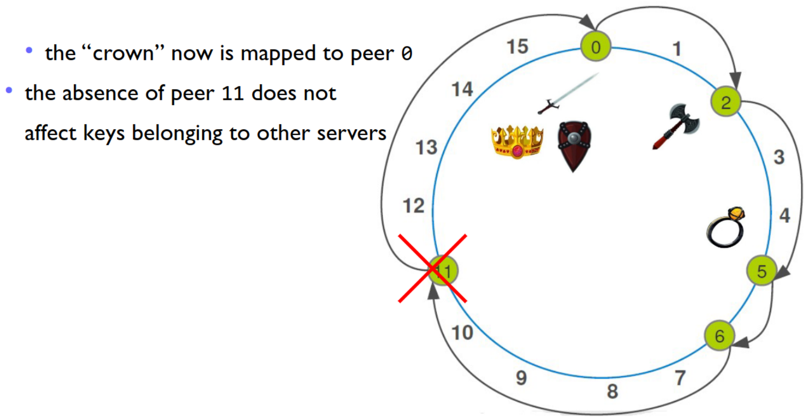


Figure 2.3: Peer 11 leaves the Network
In case a peer leaves, its keys can easily be remapped to its successor

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 Data Lookup

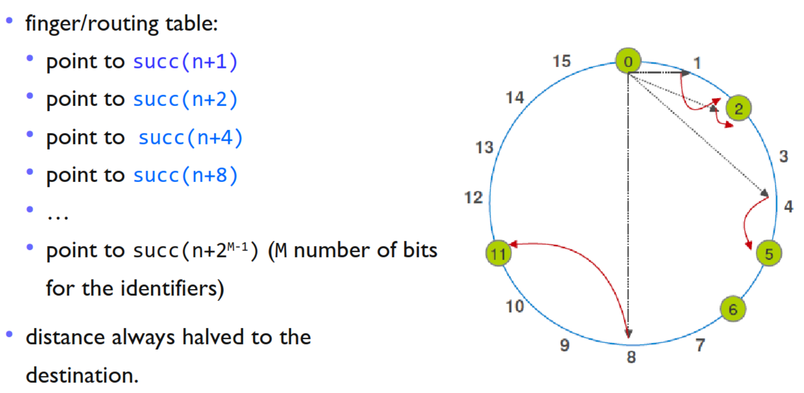


Figure 2.4: Exponential Search for DHT
The data lookup can be implemented by using exponential search, rather than performing a walk by asking each peer for its successor

Data Lookup can be sped up even more, by computing the hash $h(x)$ of the searched object, and 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 is found.

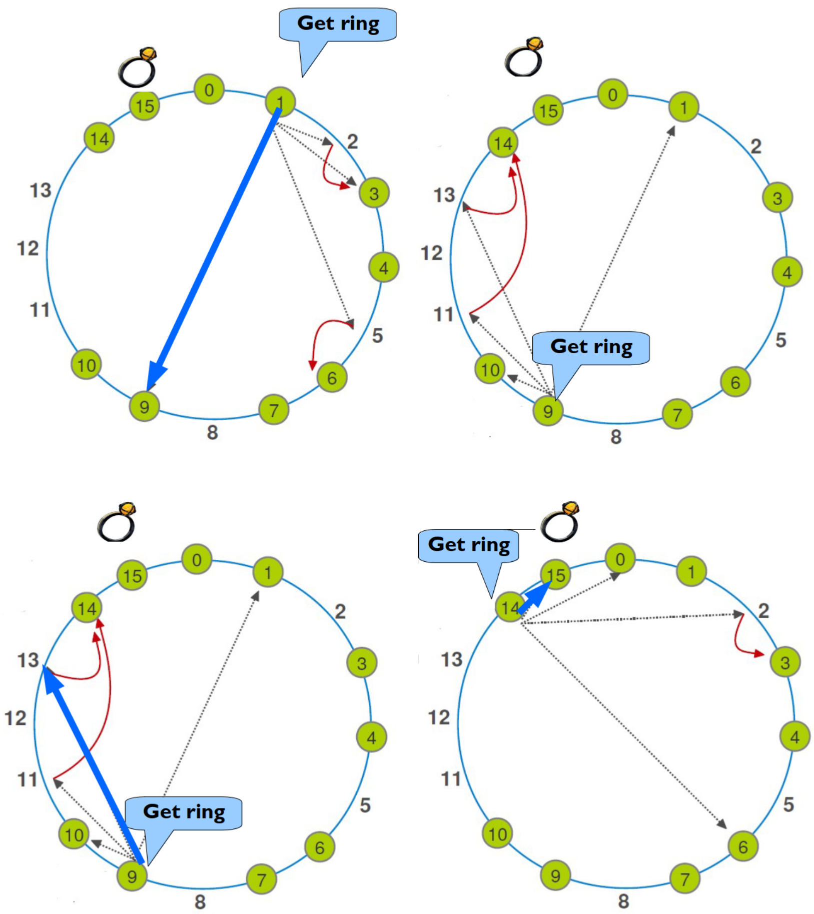


Figure 2.5: Lookup performed in the `CHORD` DHT

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.

[1]Which is found using exponential search

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

2.2.2 API, Lookup and Various Properties

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 2.6: 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
2. flexibility to send multiple requests in parallel to speed up lookups by avoiding timeout delays (parallel routing)
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 **Trie**[1] to define the logical identifier space;

Note that not all leaves correspond to nodes (peers)

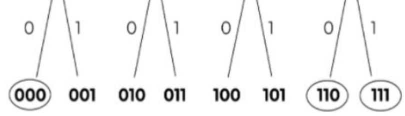


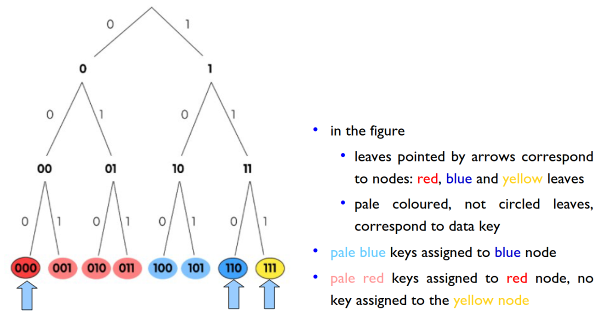
Figure 3.1: Trie

[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

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

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.

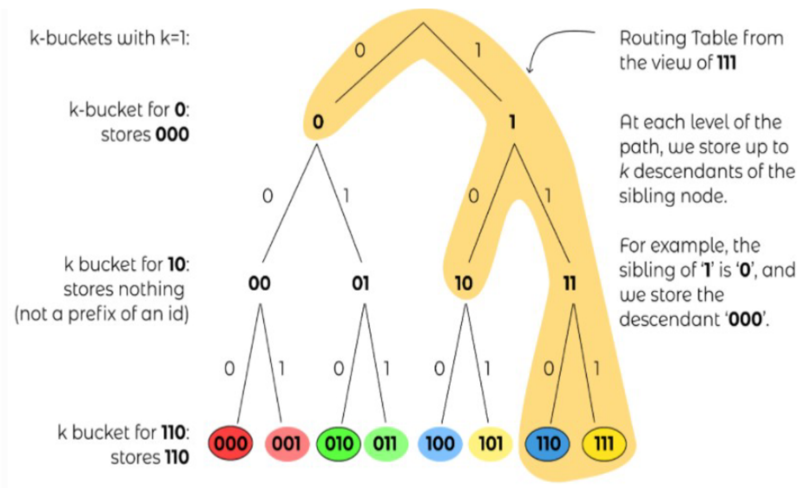
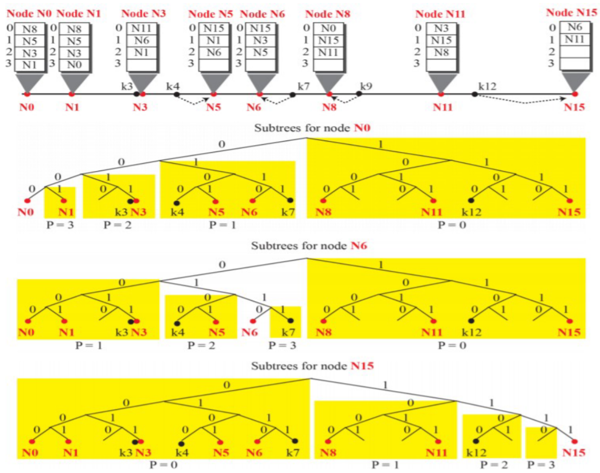


Figure 3.1: Neighbours and buckets

the rows are k-buckets ($1 \le i \le 160$)

- each one contains k contact
- each k-bucket corresponds to a subtree
- each row stores k contact:
  (ID, IP, UDP port)
- row i contains contact with distance d from the nome, $2^{i-1} \le d < 2^i$
- each entry corresponds a common prefix
- the lower the entry the longest the common prefix
- in some implementation reversed order

i		
0	$[2^0, 2^1)$	(IP address,UDP port,Node ID)$_{0,1}$ ...... (IP address,UDP port,Node ID)$_{0,k}$
1	$[2^1, 2^2)$	(IP address,UDP port,Node ID)$_{1,1}$ ...... (IP address,UDP port,Node ID)$_{1,k}$
2	$[2^2, 2^3)$	(IP address,UDP port,Node ID)$_{2,1}$ ...... (IP address,UDP port,Node ID)$_{2,k}$
...		
i	$[2^i, 2^{i+1})$	(IP address,UDP port,Node ID)$_{i,1}$ ...... (IP address,UDP port,Node ID)$_{i,k}$
...		
159	$[2^{159}, 2^{160})$	(IP address,UDP port,Node ID)$_{159,1}$ ...... (IP address,UDP port,Node ID)$_{159,k}$



Each k-bucket corresponds to a prefix and covers a subset of the identifier space: the set of all the k-buckets cover the whole identifier space. 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*.

3.4 Key Lookup

The idea for key lookup is:

1. Find the closest node to the key in your routing table via the XOR distance
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.

At each iteration, the XOR metric is reduced by $^1/_2$ and results in smaller size k-buckets

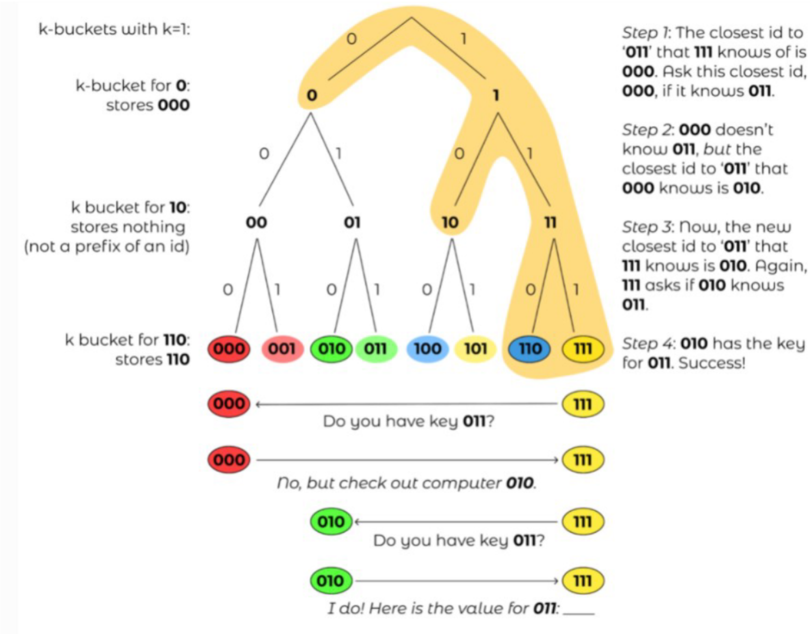


Figure 3.2: Kademlia Lookup at a glance

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`

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

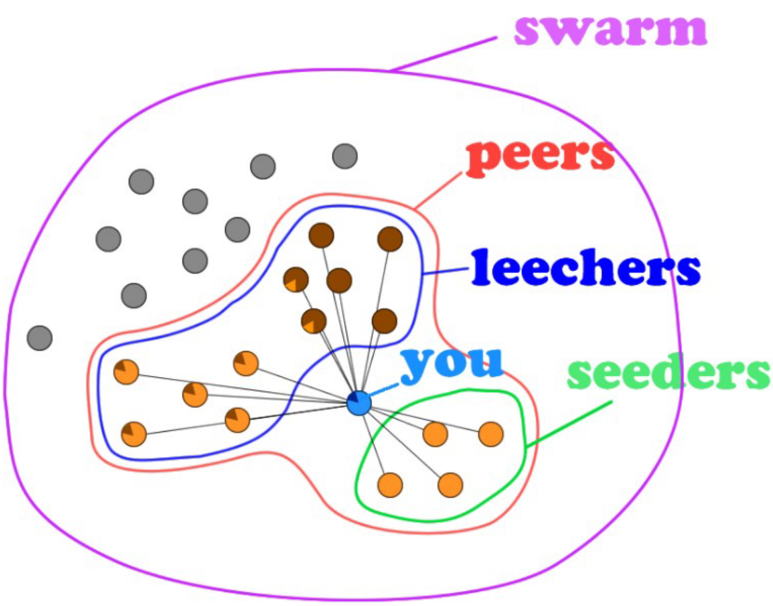


Figure 4.1: Swarm schema

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

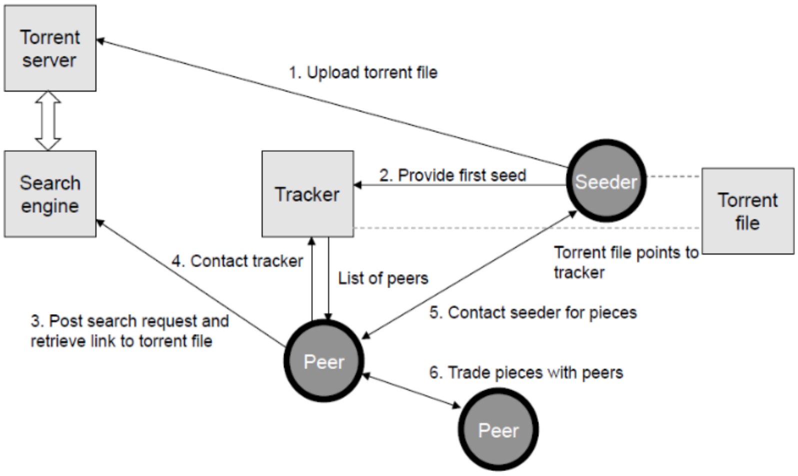


Figure 4.2: BitTorrent protocol overview
BitTorrent protocol is built on top of HTTP

Seeder

1. Upload the .torrent on a Torrent Server
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
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.

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

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
- **Random First Piece**
  Choose a random piece —only— in the bootstrap phase
- **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

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.

However, an approach to solve this problem 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.

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 chocked 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

Kademlia is the protocol used by the largest public DHTs. Bittorrent Inc. introduces its own DHT, called *Mainline DHT*. With respect to Kademlia there are 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.

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*.

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*.

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**.

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.

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.

[1] *"Libro mastro"* in italiano

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

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
   $\forall y \in Y. hard\ to\ find\ x \in X s.t. h(x) = y$
   "one-way function"
2. *Second pre-image* resistance *given* $x \in X, y = h(x). hard\ to\ find\ x' \in X s.t. h(x') = y$
   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 $2^{m/2}$ brute force computation to find a collision.

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*

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 k is chosen from a distribution with high min entropy, then it is infeasible to find x such that $H(k||x) = y$ 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
- ⋄ **DHTs**
- ⋄ **Digital Signature**
  Hash functions are widely used for public-key aymmetric algorithms, for ensuring both confidentaliaty and message integrity, 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

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

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. They are used by Ethereum, Google, and Bitcoin.

6.2.2 Merkle Hash Table

It is a data structure summarizing a big quantity of data, with the goal of verifying the correctness of the content. A **Merkle Hash Table** 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*

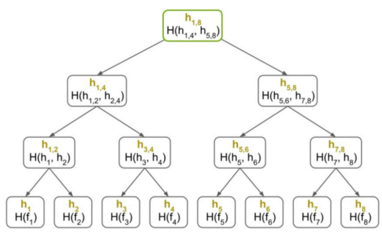


Figure 6.2: Merkle

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.

Enlightening Example - Cloud File Integrity

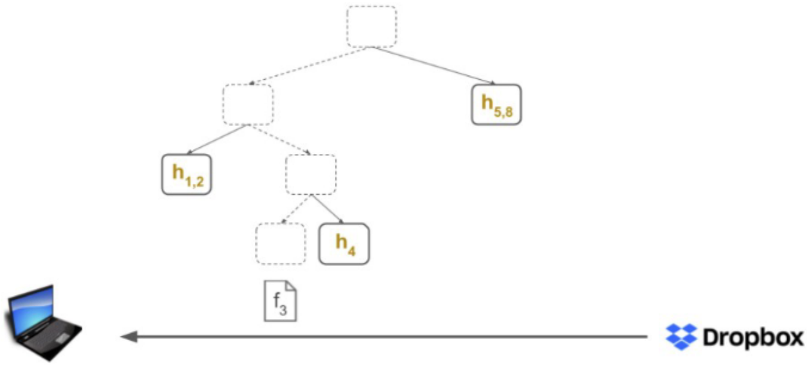


Figure 6.1: Cloud file integrity use case

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**.

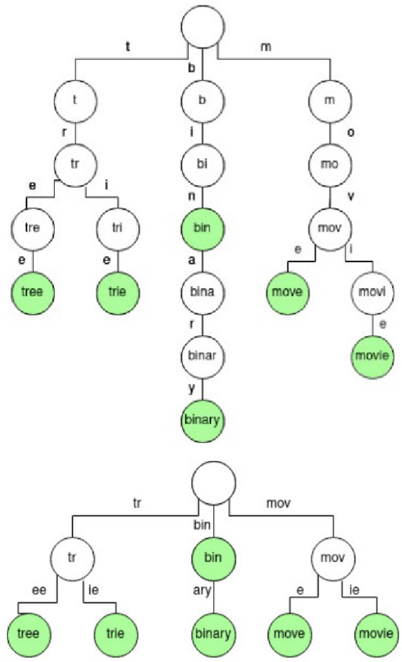


Figure 6.2: Trie and corresponding Patricia Trie

[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

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

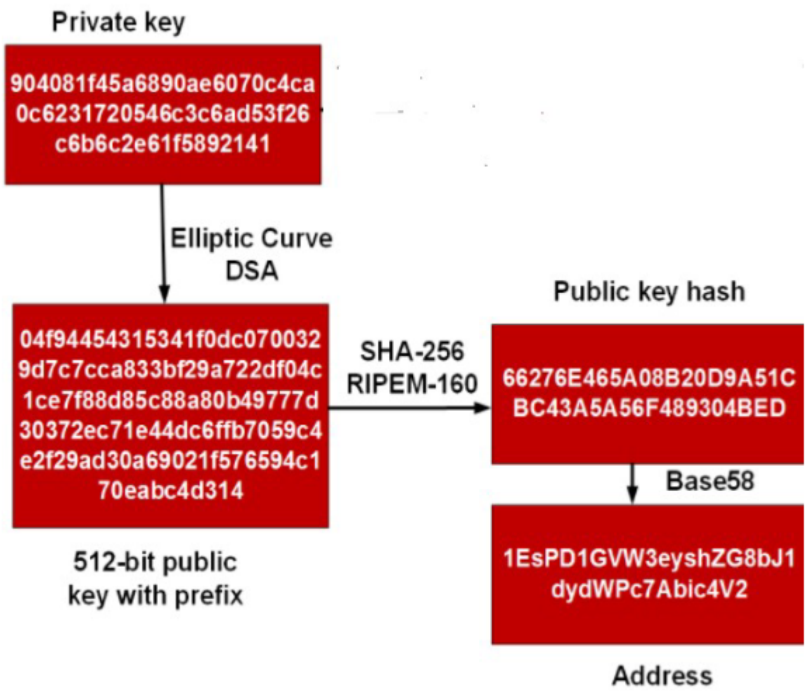


Figure 7.1: Bitcoin address generation process
`Base58` is used to eliminate from the 62 alphanumerical alphabet the characters `(0,O,l,I)` which may appear identical when displayed in certain fonts.

make inferences on who it may belong.

7.3 Bitcoin Transactions

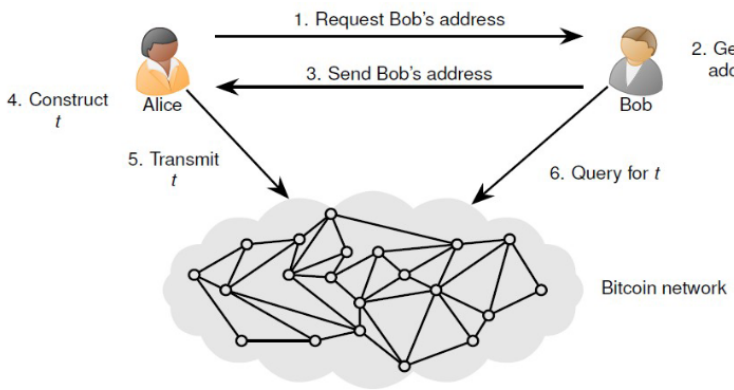


Figure 7.2: Bitcoin transaction workflow
Note that the address exchange is performed out-of-band, not through the Bitcoin P2P network

Is used to transfer funds from the sender to the receiver, and is registered on the blockchain when confirmed.

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

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.
Transactions may also be **multi input**.

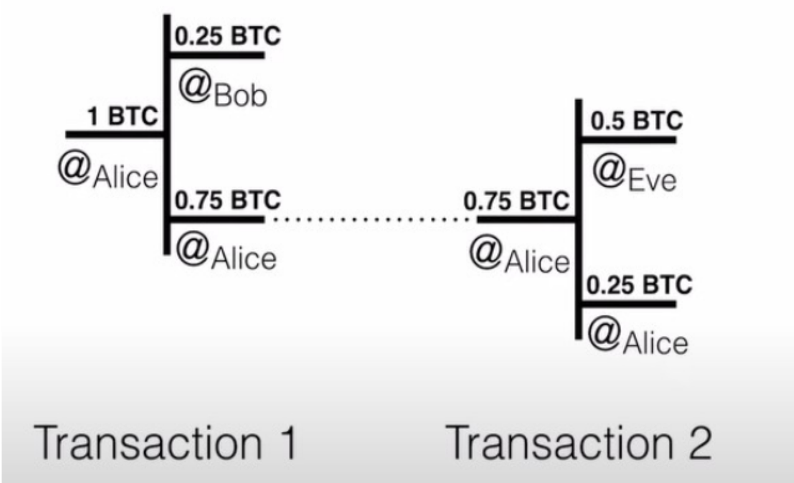


Figure 7.3: Bitcoin linked transactions

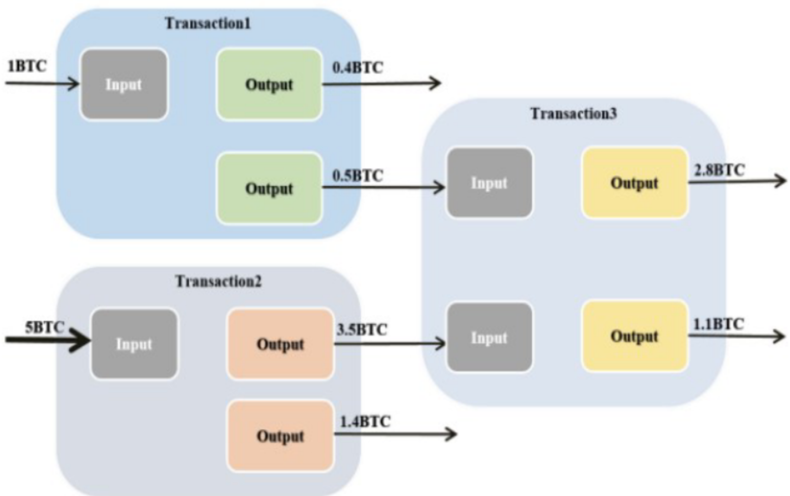


Figure 7.4: Multi input transaction

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.

Each transaction input is linked (refers) the an UTXO of a previous transaction, while each output generates a new UXTO,which is included in the user's wallet and is available to be spent. Each UXTO 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.

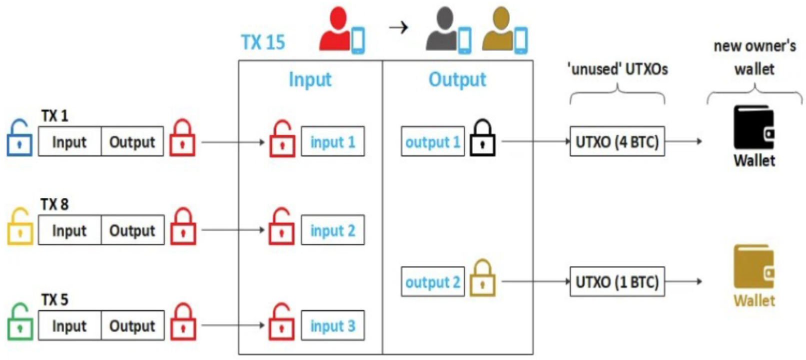


Figure 7.5: UTXO locking
Each UXTO *"locks"* the newly generated UXTO to the new owner's public key, and if they decide to use the UXTO in a new transaction, it must *"unlock"* the funds with their private key.

7.3.2 Scripts

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

The sender appends a script on top of the sent bitcoin to lock the transferred bitcoin, and the receiver[4] appends a script to unlock the bitcoin. Executing both scripts allows the receiver to spend the bitcoin.

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.

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.

[3]Not Turing-complete and overall limited to avoid endless looping and execution errors.
[4]Is it the receiver to do so?

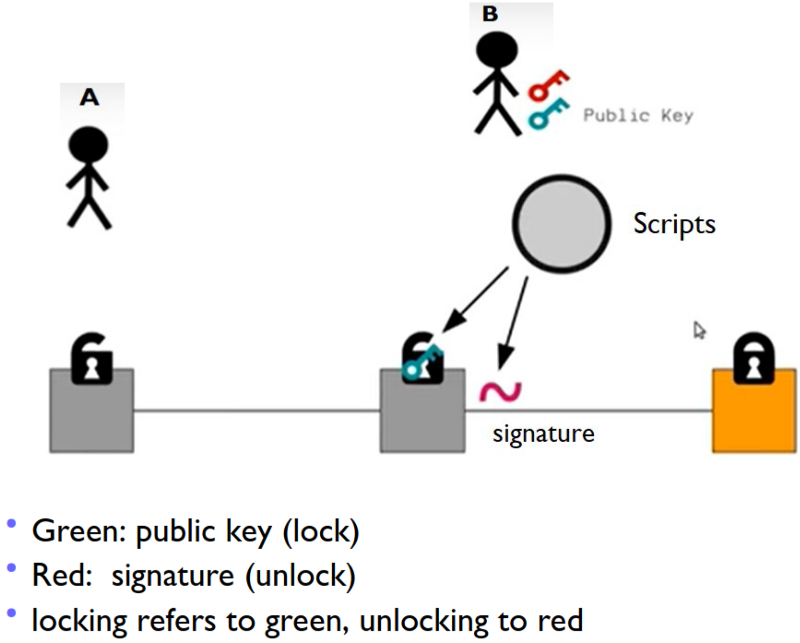


Figure 7.6: Bitcoin script example

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.
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, 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.

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
- `hashprev` The hash of the previous block
- PoW related fields:
  - Target
  - Nonce

[1] Nodes behaving maliciously

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* $F_d(c, x) \rightarrow$ `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:

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

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

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`.

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

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

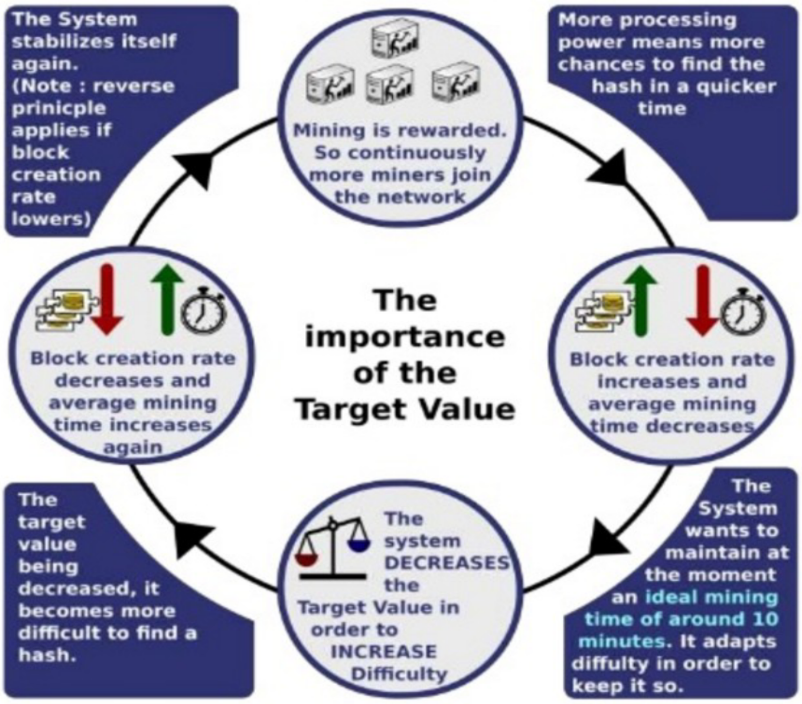


Figure 8.1: Bitcoin target cycle
A node can't avoid decreasing the target, since the other nodes would not validate its Proof of Work.

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?*

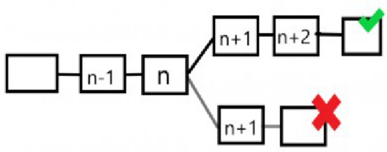


Figure 8.3: Bitcoin temporary forks

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**.*

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 is invalid.
However, there are more sophisticated attacks that can be performed on the Bitcoin network. In this chapter we will discuss some of them.

9.1 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.

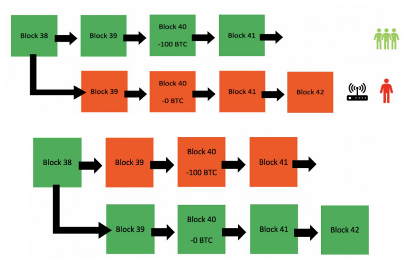


Figure 9.1: Bitcoin 51% Attack

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.