Contents

Chapter 1

Peer to Peer Systems

1.1 Introduction

The client server paradigms is charaterized by the two component:

- **Client**: runs on end-host with an on/off behaviour. Is usually a service consumer that issues requests and do not communicate directly. Surely need to known the server IP address and never communicate each other.
- **Server**: it's the service provider that receive requests and satisfy all clients requested.

Peer-to-peer systems

Differently, in **peer to peer**, we have onyl a single component: the **Peer** A **peer** runs on end-host, have a dynamic behaviour (enter/exit che network continously). The peer usually need to join a network: can be a problem because, differently from server, we don't have a fix static IP address to join or connect.
We need to connect to a node already in the net: in the scenario pictured in 1.1 we indicate it as **S** (*Server*) that is used to **boostrap** the network. It's an entrypoin to the network of peers: following to the peer's joins the server can be "detacthed" or deleted.
Peer needs also to discover each other: use a *gossip protocol* to discover and connenct two different peers: this allows the peers to provide and consume services each other, acting both as a provider and consumers based on each peer offered/requested service. There is also the need to defines **communicaton rules** to prevent, for example, *free riding* (*the problem in which we provide a service without nothing in exchange*) and incentivate participation and reciprocation.

As a first definition of P2P, we can state that: "*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***"

The resources that can be **shared** are:

- Ledgers
- Storage Space (*e.g. Distributed File System*)

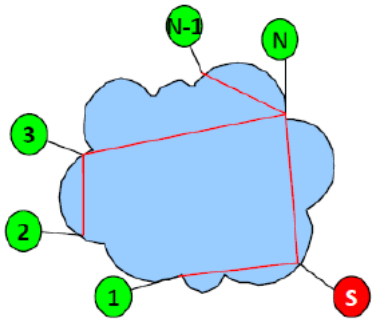


Figure 1.1: P2P Topology

- **Computing power**
- **Bandwitdth**

A more refined definition of P2P is: "*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)*"

History case: Napster

The download of the content was based on P2P but the index for search the content was centralized: the following evolutions that give birth to service like Gnutella, FastTrack, or BitTorrent have also search and content trasfer entirely completely distributed. So P2P cient acts like **servlet**: a peer behaves like a web server and the user that want to download a file behaves as a client.

1.1.1 Blockchain definitions

The **#1 definition** is: "*A shared database stored in multiple copies on computers throughout the world and is maintained without the need for a central authority (e.g. a bank, a government, Google, etc.)*"

Seems like a classical distributed database, which take to the **#2 definition**: "*replicated and consistent,* ***immutable****, append-only data storage system resistant to tampering*".
The **#3 definition** is the complete definition:
"A write-only, decentralized, state machine that is maintained by untrusted actors, secured by economic incentive in which the data cannot be deleted and cannot be shut down or censored. It supports defined operations agreed upon by participants that may not known each other and are **untrusted** but act in their best interect according to rules that incentivizes partecipation".
The blockchain store the transaction: what is exactly the semantic of this transanction depends in the nature/purpose of the application. At high level, the **basics step of the blockchain process** are the following:

1.1.2 Ethereum overview

Implement smart contracts based on protocols that use a blockckahin: they use a Turing-complete languages like *Solidity* or *Serpent*. The smart contract is executed by all nodes as the consensus algorithm is used to obtain an agreement between every node of the network on the result of the computation.

The **Ethereum blockchain** introduces smart contract that can be executed by blockchain's nodes: differently from Bitcoin where th script have only limited computational power, in Eth they can solve any

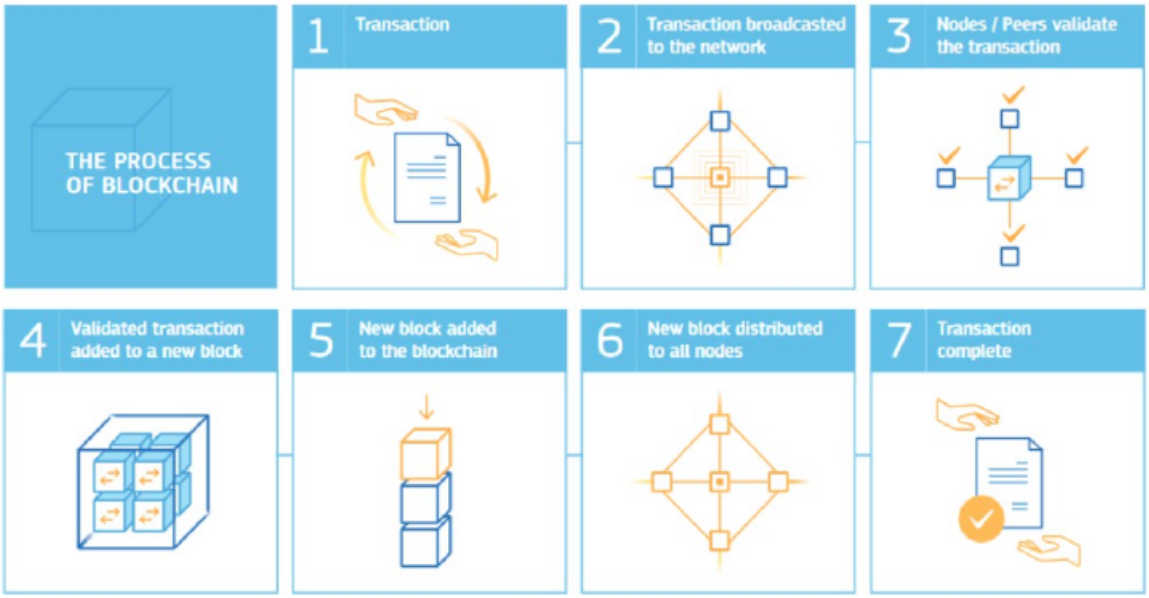


Figure 1.2: Blockchain overview working scheme

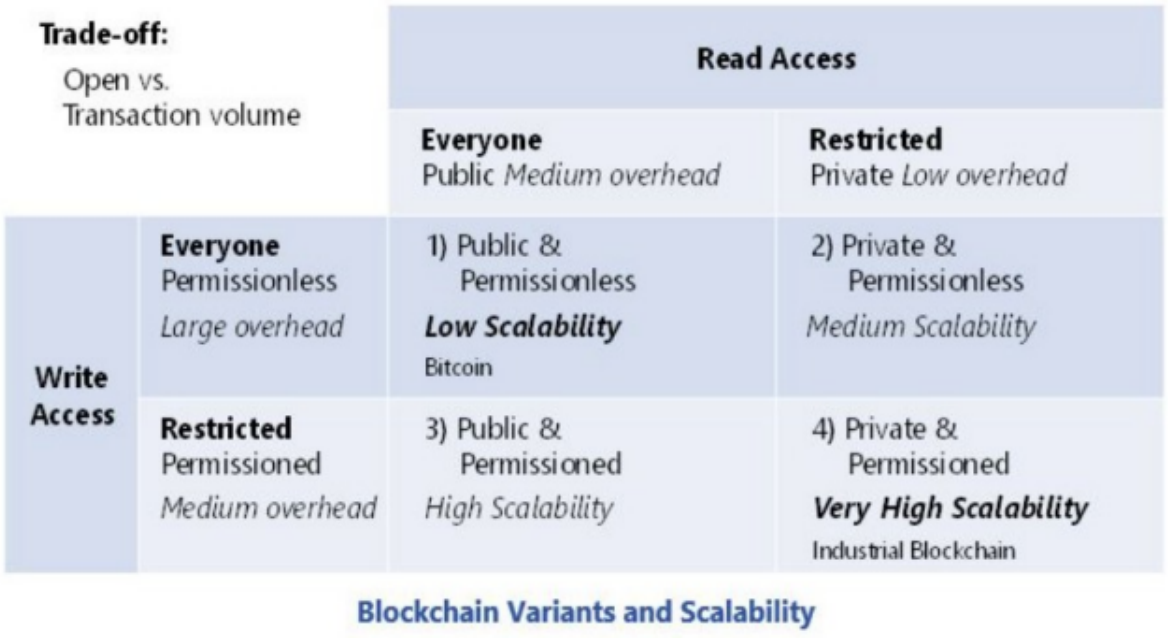


Figure 1.3: Classification of blockchains

computational problem (Turing-completeness). To avoid some sort of **Denial of Service** the concept of *gas fees* is used. See the section 4 for more information.

Generally, we can identify a **taxonomy** for the blockchains that can be charaterized along 2 axis: **Permissionless vs Permissioned** and **Public vs Private**, detailed in 1.3:

In this context, for *scalability* we means the number of transaction that can be confirmed/immutable registered in an unit of time.

To understand the **target scenario** for using a blockchain we must consider several elements. As an example, applications that require shared common, append-only database with limit capacity, based on specific necessity, as referred in 1.4:

There are some advantages on using P2P, like exploiting computation resources in excess like idle CPU cycles to have in return other resources or service from network's nodes. There is a shift of paradigms in blockchain that is concretely represented by its scientific challenges: the classic old metholodigies for the development of distributed system cannot be exploited in a context with million of nodes so classical alg/technique do not scale on networks of this size. System dynamically and of those size needs new tools like *consensus algorithms, cryptographic distributed techniques, complex system analysis tools and strategier for the peer cooperation (Nash equilibrium).*

With **blockchain trilemma** we mean a big scientific challenge in which reside the optimal equilibrium between scalability, security and high level of decentralization. Those three properties are the desiredata from those systems but are not easy to gain because, for e.g., BTC is high security and high decentralized but it's low scalable.

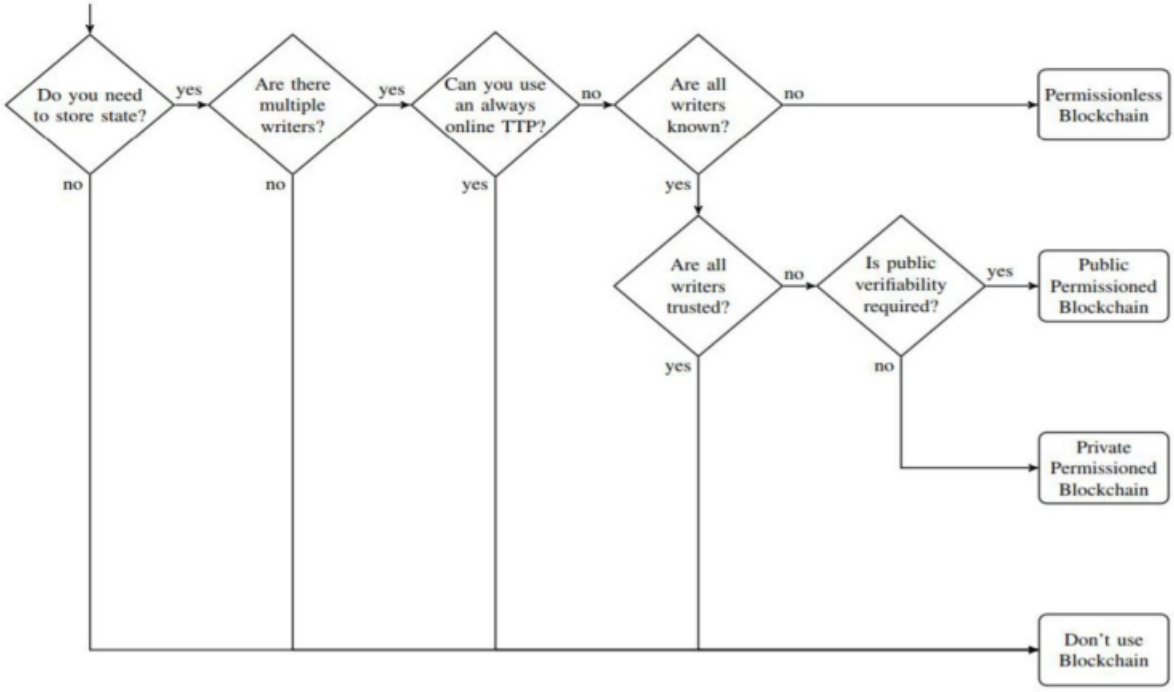


Figure 1.4: Flow to identify a target scenario

1.2 P2P Overlay Network

Napster/Gnutella overview

The basic idea of *Napster* was to outsource the storage and exchange of files to the users so most of the service was provided by users themselves. Napster was able to locate the users that can provide the file while the server was only an index of contents.

In a simplified scenario, Napster was a P2P music file sharing with a centralized index server or *Master server* with several peers: a peer get the index of contents from the master server with the address/reference to all the peers which have the desired information. There were pros of this system like the **resource sharing** in which every node pays its pariticpation by providing access to its resources. Also some cons are corcerned with the **centralization point** that still exists and were represented by the master server, which is a single point of failure.

Gnutella as Napster was a distributed, irregular network of peers without a central authority or server: the connections between the node define an **overlay network**. Peers establish **non transient direct (e.g. TCP connections) connection** between themselves: they broadcast a query throught the virtual connection of the overlay network.

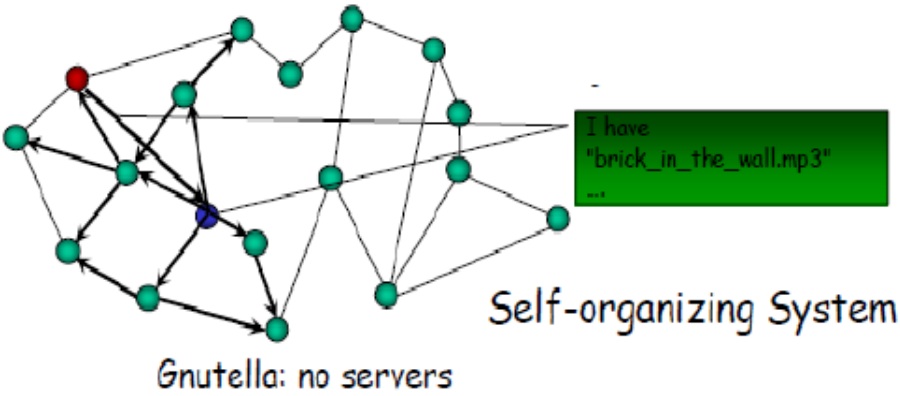


Figure 1.5: Gnutella network topology

As for Napster, the main pros was that there was no additional infrastructure, no administration and not a single point of failure but was an high network traffic with possibility of free riding.

1.2.1 Overlay Network

We define an overlay network as a *logical network* build on top of a physical network. Overlay links are **tunnels** through the underlying network: each logical link may correspond to a set of physical links and may transverse of a set of a router. (*See 1.6*).

The three main types of overlay network and an hybrid type as pictured in 1.7:

- *Unstructured overlay*
- *Structured overlay*

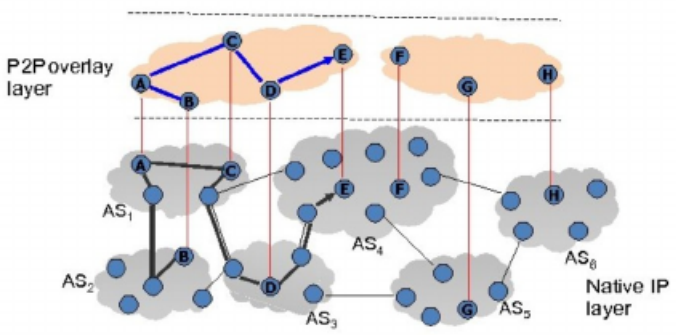


Figure 1.6: General overlay network schema

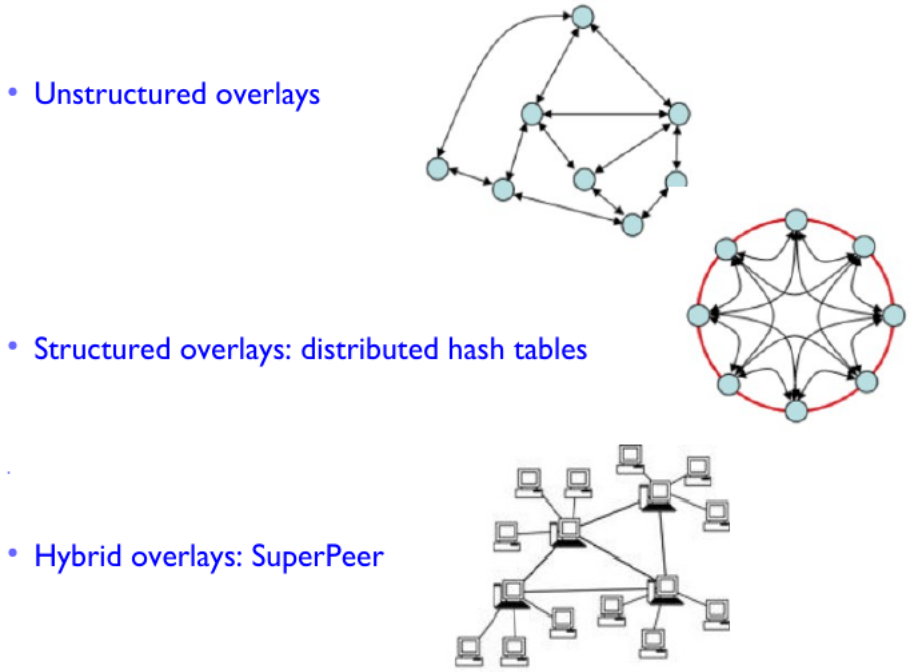


Figure 1.7: Overlay network taxonomy

- *SuperPeer*

A **P2P protocol** defines a set of messages that the peers exchange, their format and semantics and it's usually defined over the P2P overlay. All P2P protocols shares common charateristics, like:

- Define a routing strategy at application level of stack TCP/IP stack
- Identification of the peer through **unique identifiers** which are usually computed by an hash function. Particular *cryptographic hashing function* allows to reduce the collision of uuid generated.
- Charaterized by a header and payload like packets at IP level

1.2.2 Unstruncured overlays

In an **unstructured overlay network** peers are arbitrarly connected: the resulting overlay network is unstructured and make use of **look-up algorithms** by flooding the network with queries, expading the ring and use a random walk (*for example, see later*).
There are pros like *easy to code/mantain,high resiliency* and some cons like **high lookup cost** that usually are linear in n (n = *#nodes*) due to an absence of centralization authority (*like an index in Napster*) and **low scalability**. Some example of unstructured network are *Gnutella (v.04), BitTorrent (use DHT) and Bitcoin.*

Gnutella: a case of unstructured P2P overlay network The basic idea is it that propagates the query to the network, **flooding** it: there was no index information used. The connection between peers are defined **at random**: the node that requested the content choose randomly the nodes that have the desired content. The established connection for transfering content is closed after finishing so it's transient, differently from the connections used to query the nodes that are **permanent**. Gnutella protocol includes some TTL, representing a constrained on the area of network the query is propagated, limiting the node reacheable from the source: this can cause false negative but it's not reacheable from the requester.

Some general problem related to P2P network are, for example on **how to boostrap the network** or **how to find content without a central index**. The first problem on bootstrapping can be solved as pictured in 1.8: a new joining peer can be boostrapped by a third component called **repository** that provide the peer descriptors by querying known DNS servers storing the IP address of a set of stable peers (*which are always on the network*). It's also widely used an **intenal cache** mechanism in which each client stores the IP address of peers contacted in the current and previous sessions: the cache can

be dynamically update by gossiping with neighbour peers to drop addresses of nodes that exited the network.

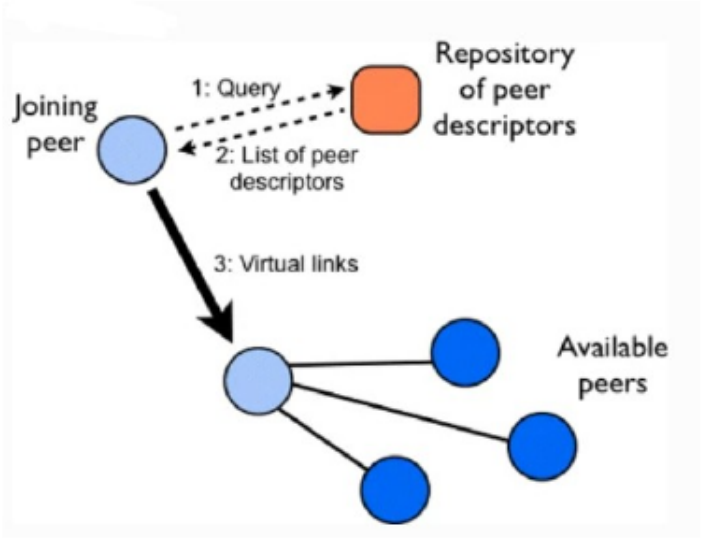


Figure 1.8: Bootstrapping nodes without authority centralization

Let's deep dive into a **Network Discovery** for an *unstructured P2P Network* in which the network topology is tipycally random. The **Network Discovery** follows those steps:

- **Step 0**: join the network
- **Step 1**: determine "who is on the network" by "ping" message to announce your presence on the network so other peers can respond with a "pong" message and also forward your "ping" to other already connected peers. A pong packet also contains info on the peer sending it, allowing two not-directly connected peers to communicate.
- **Step 2**: after established a connection with a set of peers, the searching is possible by sending a "*query*" that asks other peer for some content: an example of a query packet "do you have any content that matches the string "*Back to Black?*"". In unstructured network, the querying is executed by broadcasting, implementing a sort of **flooding**. By receiving the query peers check to see if they have matches and respond (if they have matches), otherwise send packet to connected peers if not to verify if other peers can answer the given query.
- **Step 3**: downloading when the desired resource is founded. The content transfer use **direct connections using HTTP's GET method**.

The described steps are applied to the scenario pictured in 1.9: the schema refers the *ping-pong* protocol in *Gnutella*. In this schema the *TTL - Time To Live* is used to limit flooding: when TTL is equal to 0 is discarded.

.

Searching by flooding

For the first case, consider the scenario pictured in 1.10. The connection between the peers (*black one*) are **stable** while the red one is only used for transfer content through an HTTP connection. Finally, the **blue connections** are used to forward query to their neighbours.

In the pictured scenario of 1.11 scenario, red node are searching for red content and yellow node are searching for yellow content: usually there are multiple copies of content for redundancy purpose. By **flooding**, as pictured in 1.12, they are able to detect some of the content they're interested: some of the copies are not finded due to the use of TTL tgat limits the number of nodes to query.

When the content is found on a node, the protocol allows to implement *backward routing* to the original node that executed the request and carried the information to connect directly to the node and transfer the content. This poses a problem due to *backward routing* mechanism: each node needs to mantain a table and usually an huge number of connesions on the requester (*node S*) is not suitable for scalable networks.

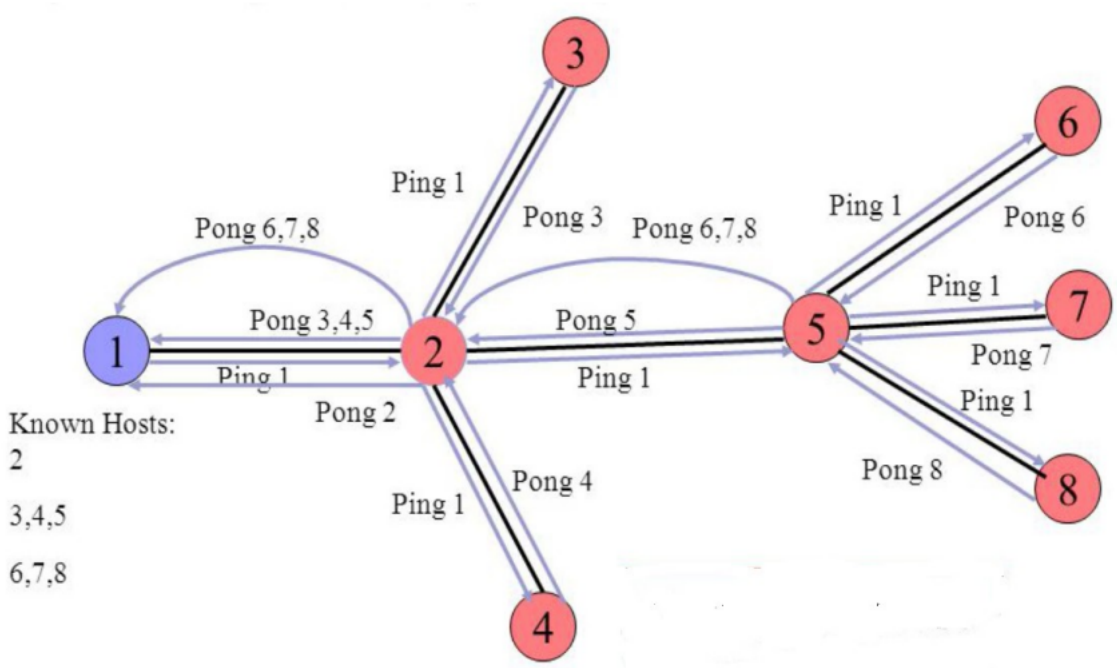


Figure 1.9: Broadcasting request by flooding

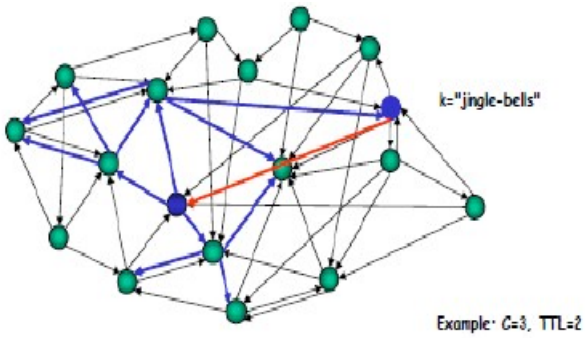


Figure 1.10: Distinction of connection

The use of TTL can also result in **false negatives** due to the limit on the number of hops setted by TTL: this also reduce scalability on the entire network. So **backward routing** is used to reply messagge through the *non-transient* connections.
Making a parallelism with a *bitcoin transaction*, we have to connect (or we are) a full node (enabled and with the full blockchain data) and to add a transaction, perform the *flooding* to all nodes to propagate this transaction in the P2P net underlying the blockchain.
Many popular P2P network are *unstructued* and use **flooding**: so overlay is used for content detection and use a direct HTTP connection for content download.

Expanding ring/Iterative deepening

It's a method to avoid TTL to limit and possible not finding the content: the key diea is to use a **sequence of flood with increasing TTL**. Follows those steps:

1. Choose a ratio of the neighbors (*random subset*) or all of them
2. Start a BFS with a low TTL value
3. If the search is not successuffl, repeat BFS at increasing depth by increasing TTL

Random walk / Drunkard's walk

The key idea is to build a path by taking succesive steps in random directions: the path is described by a *Markov chain* so previous states are irrelevant for predicting the subsequent state. The probability of

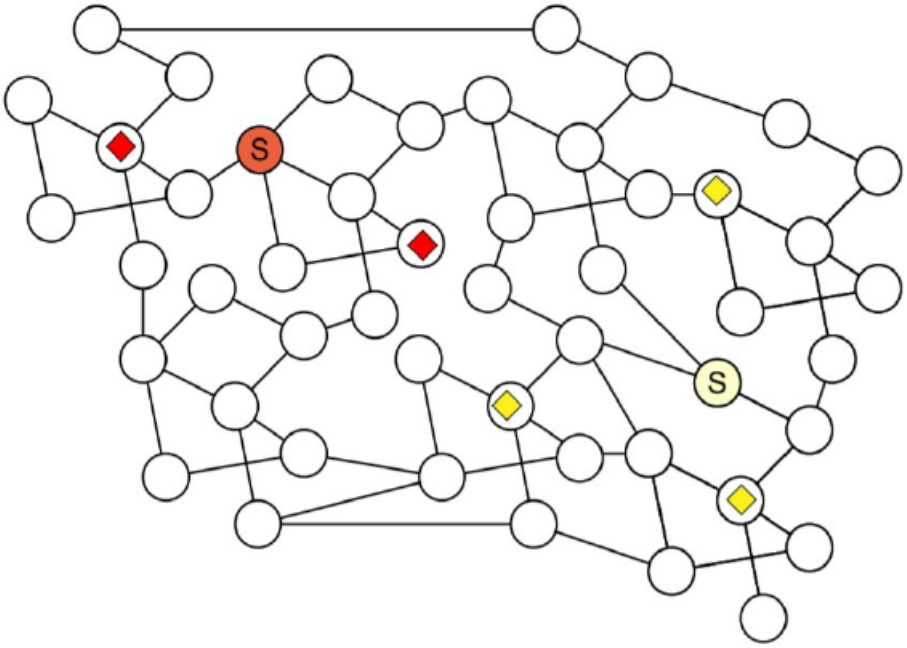


Figure 1.11: Base scenario: Red node search red content, same for yellow

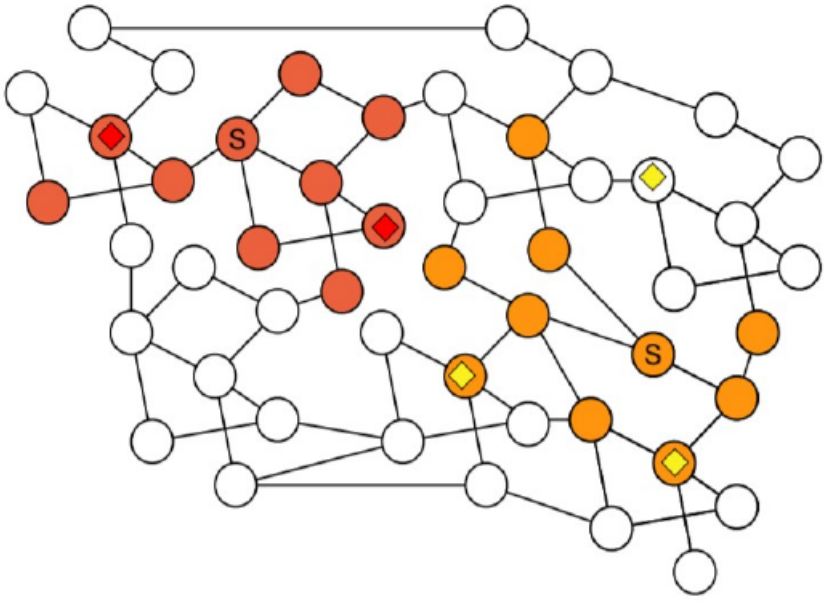


Figure 1.12: Node visited after flooding

distribution of the future state is a function of the present state alone.
Another version if based on sending out **k query message** to an equal number of randomly chosen neighbors: is called **K-Random Walk**. Each step follows each own path by choosing one neighboard to forward it and still the TTL s used to stop the search. Each path is called a **walker**.

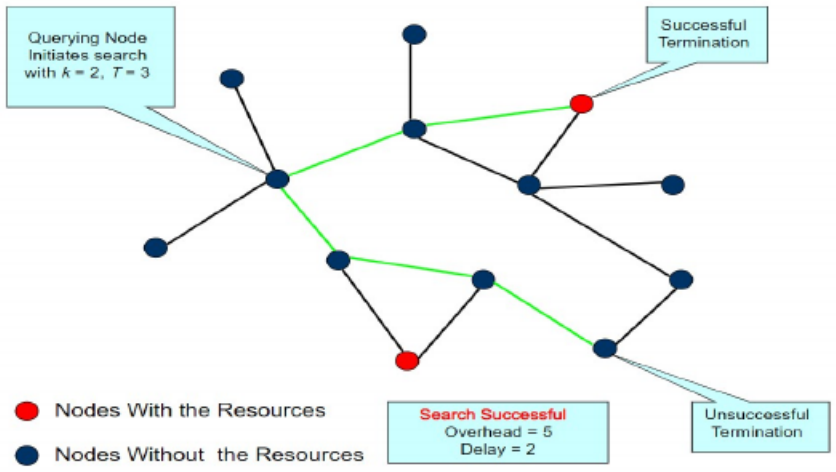


Figure 1.13: Drunkard's walk example

The routing is a **content-based routing** so the searching is driven by particular hash data structure (*See DHT*). There are two main methods to *terminate each walker*:

- Based on TTL
- Checking methods like walkers periodcailly check with the query source if the stop condition has been met. There are approaches in which the nodes are not choosen randomly but probabilistic information are used to direct to high degree neighbor: this allows to tune the probability yo choose a neighbor and drive the searching to *good* neighbos (*the one with the desired content*).

1.2.3 Structured P2P overlay

In this type of overlay network, the choice of neighbors is defined according to a given criteria so the resulting overlay is **structured**.
This approach tends to guarantee *scalability*: it use a *key-based* loopkup, the network structure guarantees that the lookup of information has a complexity of $log(n)$ and this complxity is also guaranteed for peers joins.

Hierarchical overlays

We have two different of peers: **peers** and **super-peers**. The simple peers are connected to super-peers, super-peers are connected each other: they usually have high bandwith and better performance respect to simple peers. Super-peers are used for indexing the resources and usually the flooding is restricted only to them, so too the content trasnfer was limited between super-peers.
The revolutionary part of this schema in Gnutella and Skype is that the **election** of super peer was automatic: in Gnutella there was a concept of *self-promotion*, in Skype ultrapeers are statically defined.

Overlay classification summary

As pictured in 1.14, in *DHT - Distribute Hash Table* the complexities are represented by mantaining the fixed structure, considering also the join/leave of the peers that can give more churn an lose the given fixed structure and the theoretical cost of operations.

1.3 Distribute Hash Tables: Chord

The problem of retrieve content in P2P network can be summarized by asking where the information should be stoed and how to find it without a centralized server. In this scenario, the action to carried out are two:

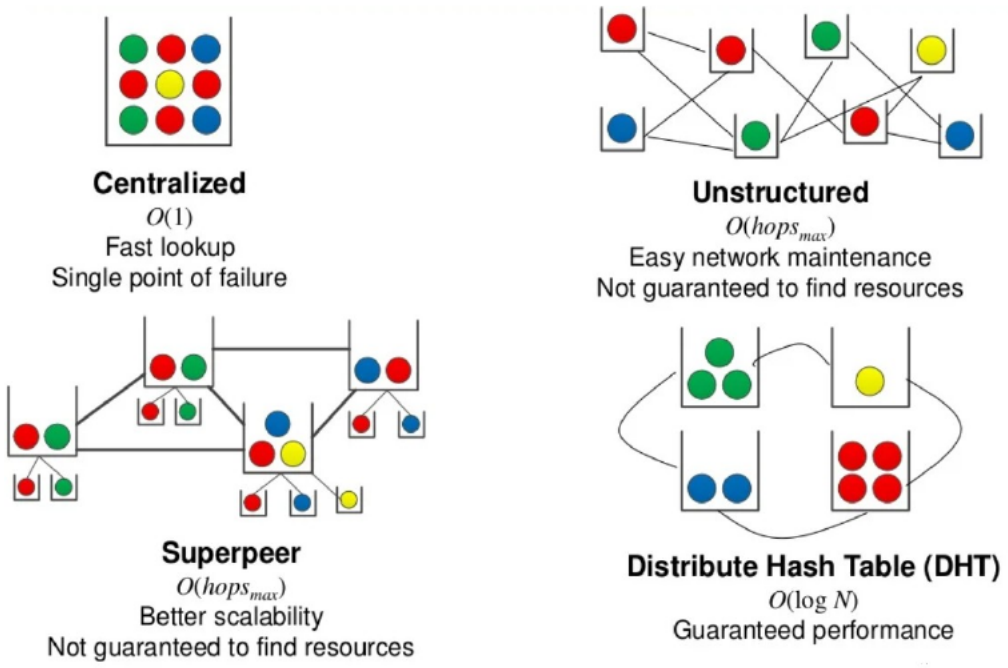


Figure 1.14: Overlay classification summary

- **Searching**: search guided by the value of a set of attributes of the content
- **Addressing**: pair **unique identifier ID** to the content and use ID to retrieve it. Usually the **id** is the **hash of the content** obtained from applying a cryptographic function (**e.g. SHA**) that return a **fixed-length identifier**. Surely the pros are the efficient object detection, the cons are the complexities of mantaining the addressing structure. In this scenario, those two action allows to drive towards a distributed network content like **IPFS**.

The *motivation* that pushed DHT are implied by the following two different way to find a solution to the previous two points of *searching* and *addressing* the content:

- **Centralized approach**: a server that index the data. The search time will be $O(1)$ and the spare required is $O(N)$ with $N = Amount\ of\ shared\ content$ and also complex query are easily managed.
- **Fully Distributed Approach**: in an unstructured network, the **search worst-case** is $O(N^2)$ because each node contact each of its neighbours. Despite this, optimizations can be done by using **TTL or identifiers to avoid cyclic paths**, obtaining a serach cost of $O(N)$. Also the space required is $O(1)$ because does not depend on the number of nodes in the system so no data structure to route queries (**by flooding**).

The key idea behing DHT is to use an **hash table** that have a $O(log(n))$ number of entries and allows to retrieve data in $O(log(n))$, avoid false negative and it's suitable for self-organizing systems with an high dynamic of node that leave/join. A sketch of the previous evaluation is pictured in 1.15.

Memcached

Memcached is a distributed memory-object caching system for dynamic web caching: a pool of caching server to provide fast access to information. This allows to reduce database server load so acces to the database only on cache misses. It's implemented by splitting an hash table in serveral parts hosted by differents server so to bypass memory limitation of a single computer.

1.3.1 Scaling out - Distribute Hashing

The key idea is split the hash table into several parts and distribute it to several servers: use **hash of resources (or URLs of resources)** allows to map them to a dynamically changing set of web caches. So the URL's **hash** is the key to access the content and the chosen cryptographic function allow to map

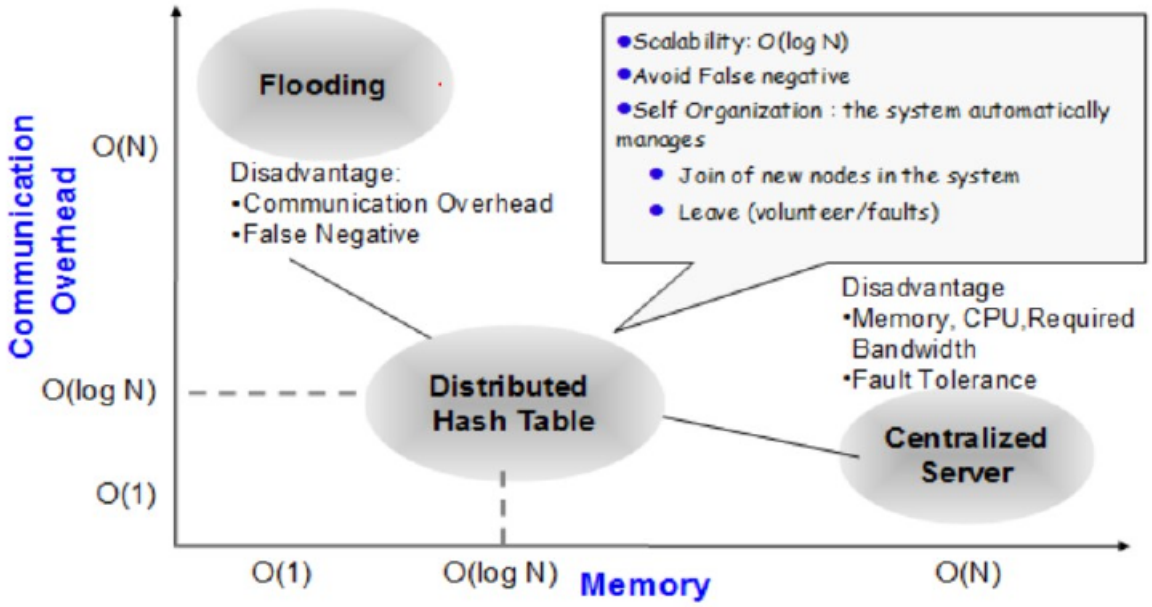


Figure 1.15: DHT rationale: trade-off between communication overhead and scalability

the key to a single server. Each machine (user) can locally compute which cache should contain the required resource, referenced by an URL (**this avoid inter-cache communication**).

The **rehashing problem** describe how manage the system in case of distributed hash table: in case failure of one server the content of this server is lost so it's not suitable for large data chunk that need to be moved or **remapped** each failure.
Let's make an example: suppose to have as cryptographic hash function a classical hash function that support 4 caching nodes. Store the resource with URL x at the cache, implies calculate:

$$SHA(x) \rightarrow ID\text{-}160\ bit\ \%\ 4 \rightarrow Cache\ ID$$

Now, if the storage gets bigger so suppose to add more 2 caches: with 6 caches, need to recalculate where all the URLs are stored. The only URL stored on the same node as before are those were:

$$SHA(URL)\ mod\ 4 == SHA(URL)\ mod\ 6$$

so with 10 buckets, and 1000 keys abot 99% percent of keys have to be remapped and this implies an huge data traffic load on nodes.
To partially solve this problem, we can evaluate the **consistent hashing** method. It's an hash techniques that guarantees that adding/remove nodes allows to remove/move a small percentange of full data.
The idea is to **map a contigous interval of hash values** to the same node, not a set of sparse value. To map an interval to a node, in addition to hashing the name of the object (**whenever they're URLs or IP addresses**), it's necessary to also hash the names of the nodes in same space (**same space means to the same co-domain; the identifier of a node is obtained by hashing their IP addresses**). The distribution scheme does not depend directly on the number of servers: each node manages an **interval of consecutive hash keys** and not a set of sparse keys. Intervals are **joined/splitted** when nodes join/leave the network and key ridistributed between adjacent peers.

Chord: construct a DHT

Chord is a DHT which uses a **logical ring** as the data structure to distribute the contents. Use a logical name space, called **identifier name space** consistiting of identifiers $0, 1, 2, , ...n$: they are defined as a logical ring **modulo** n. Every node picks a **random identifier** through **hash** H, used both for nodes and content.

The picture in 1.16 shows how nodes are mapped in the ring: each node also has a **successor** to it thus the successor can be defined via function `succ(NUMBER)`. The parameter `NUMBER` belongs to the hashes set. The successor is the next hash in **clockwise direction.**

The actual content is mapped to the ring but in case the hash does not correspond to a node, it is mapped to the node **subsequent** to it 1.17a. As one node is **removed from the network, its content is remapped to its successor**, as showed in 1.17b.

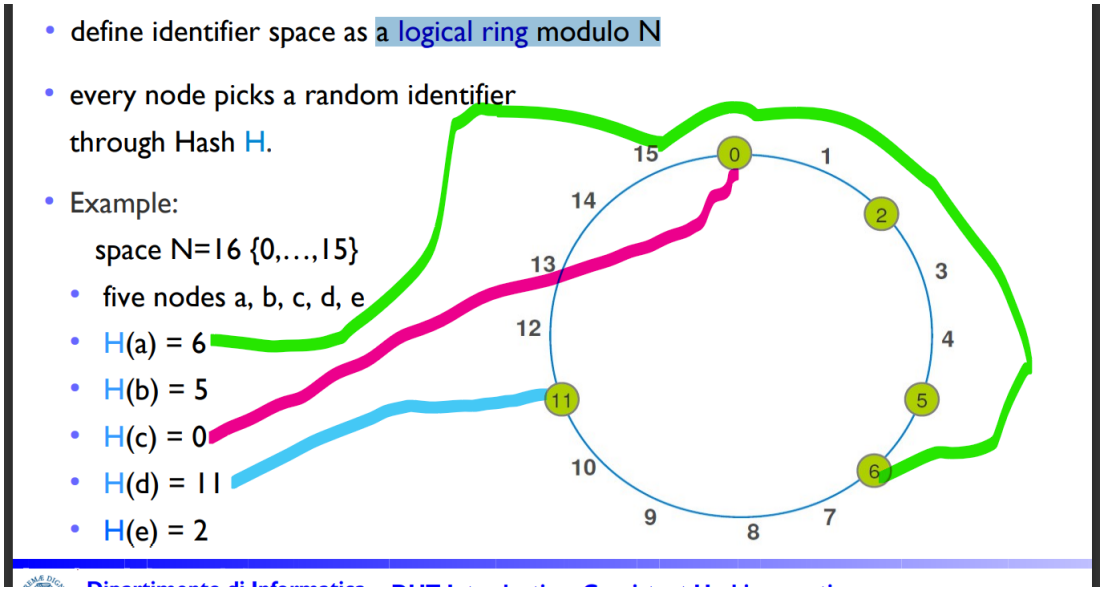
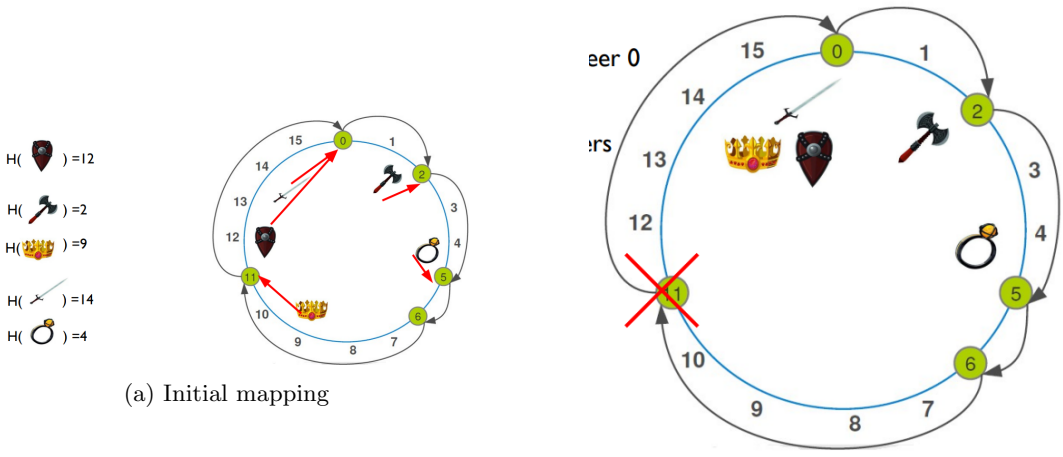


Figure 1.16: Chord logical ring

.



(a) Initial mapping

(b) Remapping the content

Figure 1.17: Chord logical ring

Instead of remapping all the keys, only the key of a leaving node are remapped so having n number of servers and k number of keys, the ones to be remapped are k/n on average.

1.3.2 Node Join

As for the **introduction of the new nodes**, inserting a new node brings the **issue of remapping some of the content hashed between its successor and predecessor**. The following figures show the crown moving from 11 to 10, as 10 joins the network. As the successor of 9 now becomes 10 instead of 11.

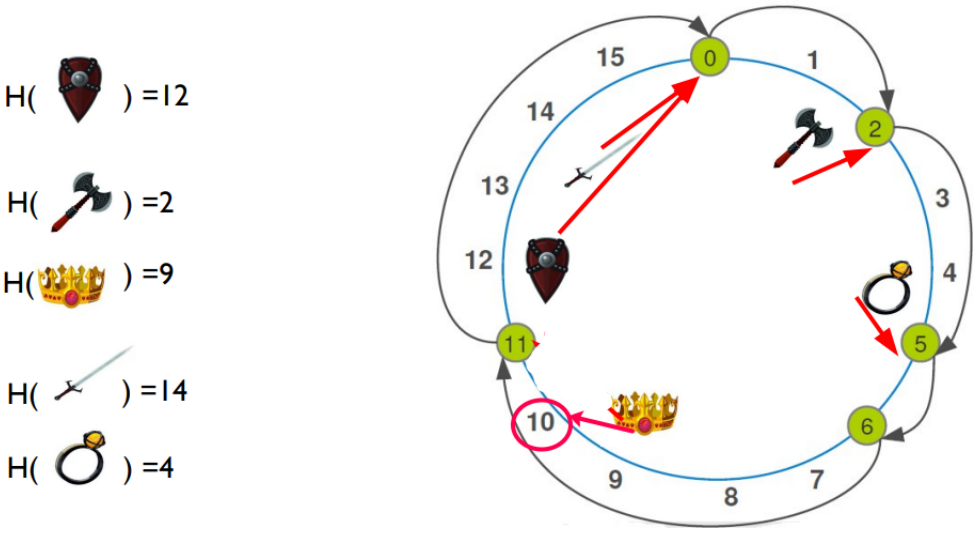


Figure 1.18: Joining node sketch

There is the distinction of two types of leaves from the network:

- **Voluntary leave**: in this case all the content stored on the node is redistributed. All the other nodes remove it from their routing table.
- **Node failure**: to prevent any content loss that can happen in this case, there is the use of **data replication** to add redundancy to the ring. The routing tables are updated periodically by exploring the routing paths and finding offline nodes.

In a **voluntary leave**, for a node there is the partitioning of its address space to the neighbouring nodes.

1.3.3 Lookup

Lookup for nodes only connected to one node (**who is its**) successor in the **routing table** is seen in the pseudocode pictured in 1.19.
The lookup is recursive, the nodes which the query is forwarded do not return the next node to lookup up. Instead they will recursively search for the originator. The result is returned at the end of the process to the originator. The originator does not control the query flow. To find the *successor* of an *ID* the *predocessor* must be founded first. This means entails travelling the logical circle. The *predocessor* is the first **non-NULL node in the ring** which as an ID associated. The algorithm to find the successor of a node, given the ID is sketched in 1.19:

Instead of connecting only to the successor, a node could store connection with other nodes in the interval of $(n+1, n+2, n+4, ..., n+2^{M-1})$ where:

- M is the routing table size (**node table**)
- N is the modulo
- N ans M are in a relation of $N = 2^M \leq M = log(N)$ which is the number of the entries in the routing table So, in the following picture, starting from node 0 we can compute *succ(n+1), succ(n+2), succ(n+4), succ(n+8) ... succ*$(n+2^{M-1})$. So for all i in range 1..M every node known *successor*$(n+2^{i-1})$. The key idea is pictured in 1.20

```
// ask node n to find the successor of id
procedure n.findSuccessor(id) {
   if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
   else if (id ∈ (n, successor]) then
      return successor
   else // forward the query around the circle
      return successor.findSuccessor(id)
}
```

Figure 1.19: Pseudo-code to find the successor

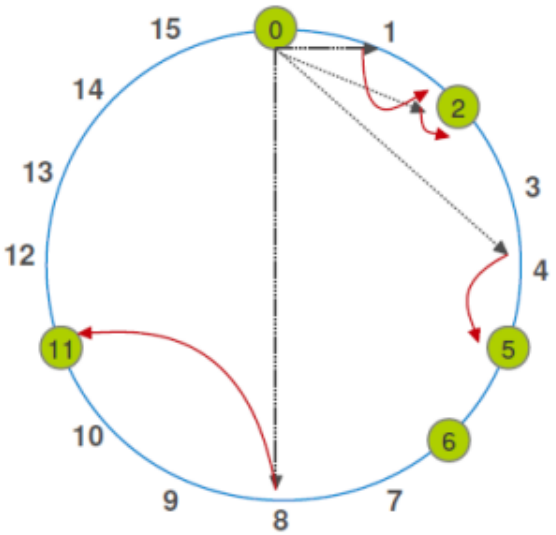


Figure 1.20: Node connect with other nodes in interval $(n+1, n+2, n+4, ..., n+2^{M-1})$

The particularity of the system is how it searches for the successors if the current does not store the content: the nodes in the routing table of a node could be all the nodes (**but not likely, as it requires space**) or just the successor node. The latter case has a complexity of $O(N)$ to find a content, having N nodes: its use can be seen in the following snippet 1.21 where instead of only asking for the *successor* of a node, we also search for the *closestPrecedingNode* to **forward the query around the circle**.

Finger(i) in the previous snippet is the **hash** of the node identified by i. The value computed is used to find the closest preceding node to the current node. Given that this is a blind area for the current node: this allows to find a node when the predocessor considered at first is NIL.

An example The example in 1.22 shows the nodes in the routing table of a node and the lookup of an item on node 15. The table has size $N = 16$, which means than $M = 4$. Node 1 has $ID_1 = 1$ and the routing tables entries as: $2(ID1 + 2^0), 3(ID1 + 2^1), 5(ID1 + 2^2), 9(ID1 + 2^3)$**.**

```
// ask node n to find the successor of id
procedure n.findSuccessor(id) {
  if (predecessor ≠ nil and id ∈ (predecessor, n]) then return n
  else if (id ∈ (n, successor]) then
    return successor
  else { // forward the query around the circle
    m := closestPrecedingNode(id)
    return m.findSuccessor(id)
  }
}
```

```
// search locally for the highest predecessor of id
procedure closestPrecedingNode(id) {
  for i = m downto 1 do {
    if (finger[i] ∈ (n, id)) then
      return finger[i]
  }
  return n
}
```

Figure 1.21: Pseudo-code to search also for `closestPrecedingNode`

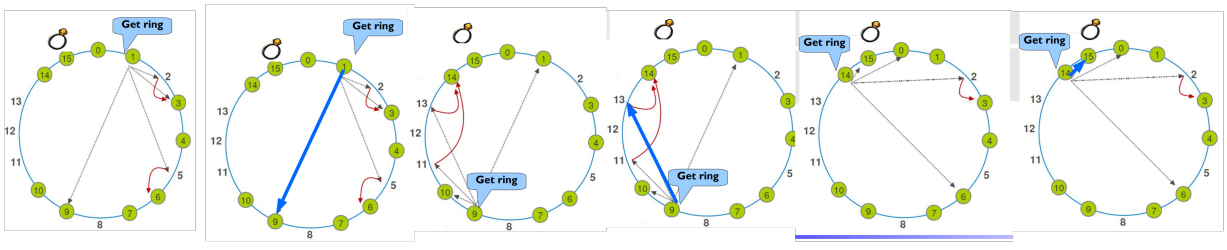


Figure 1.22: Chord Ring operations example

1.4 Kademlia DHT

Summarizing the Chord DHT, we have a virtual space address: we assign the keys to the first node encountered going clockwise starting from key. The keys are represented with the same **color** of the node they are mapped to. This colors forms a partitions, as showed here:

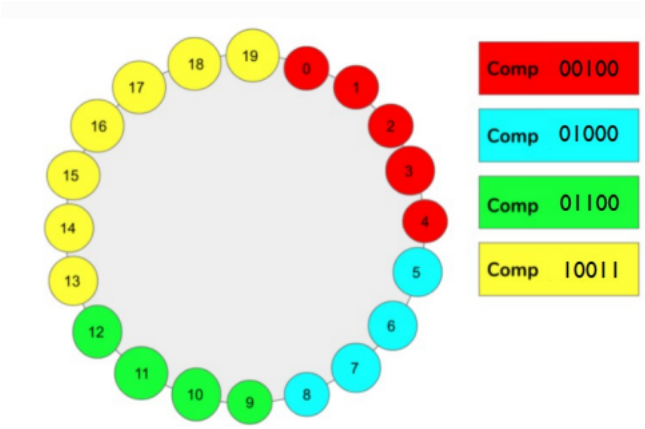


Figure 1.23: Chord DHT simplified

Kademlia uses a different approach. It represent the *de facto standard* searching algorithm for P2P networks on the internet. It's a protocol spefification for storing and retrieving data across P2P network charaterized by:

- *Decentralized*: data is not stoed on a central server, but rather dedundantly stored on peers
- *Fault Tolerant:* if one or more peers drops out of the network, the data is stored on multiple peers so should still be retrievable
- *Easy storing technique*: data is stored in key-value pairs

Respect to other DHTs, Kademlia have success because in query routing, the propagation of the query allows the peers to *enrich* their routing query. So propagating the query allows to add information on a peers table in a symmetric way. Also, the querying method allows to send the query in **parallel** trhough different paths. Kademlia also uses **iterative routing** so the peers that submitted the query is able to control the query ad each steps (*differently from recursive routing*).

Structure of identifier space The identifiers of node and data are organized in a **complete binary trie**: it comprises a *k-ary search tree or a prefix tree, allowing locate specific keys within a set.* In the *trie*, edge are annotated with binary digits: a *path* identifies a key whose value may be stored in the leaf. The general idea is skected in **??**.

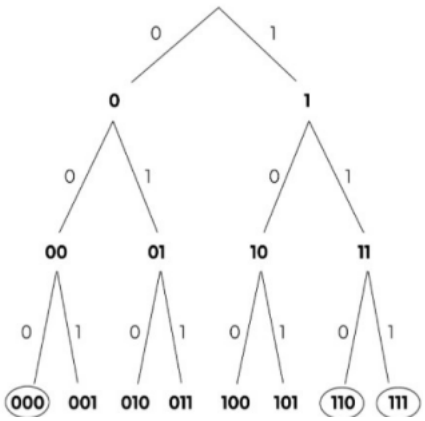


Figure 1.24: Structure of identifier space with 3 level trie

The logical identifier space is defines by the leaves of the tree, so nodes and content have identifiers in the leaves. In the image, the circled binary corresponds to the nodes. Nodes are positioned as leaves

in the trie (*sure not all leaves corresponds to peers*). It's necessary to define a rule to partition the keys (*contnet*) among the nodes: this should also respect the rules of consistent hashing. A new node entering the network now is part of the trie but each node does not store the entire trie on its own but only a subset of them that allows to guarantee a logarithmic cost of operations.

Mapping keys to nodes A key is assigned to the node with the *lower common ancestor*: first find the **longest prefix** between the key and the node identifier and assign the key to that node, as sketched in 1.25.

In the figure 1.25, leaves are pointed by the vertical arrows and corresponds the nodes. The pale colored leaves are assigned to the circled nodes. In the example, the red leaves are in common *0* so the assigned peer is *000*, in blue despite *100* and *101* shared *10*, they are assigned to peer *110*: this is an arbitrary choice becayse they only shared with the two peers in the same subtree only *1*, in this case the choice was *110* but also can be *111*. This *arbitrary property* can be exploit to alleviate a known problem in DHT about balancing the workload among peers: despite the using of cryptographic hash function, in a scenario with popular content highly requested and other content never requested, this can create a workload balancing problem.

If two computers tie for the longest common prefix, they must look to the *Most signicant bit (MSB)* where they differ, of index b so assign the key to the node whose id bit b equals bit b of the key. As an example, consider key 100 and nodes ids 110 and 111. The keys and the node ids share a common prefix of 1. The node ids 110 and 111 differ in the last bit so the last bit of the key is 0: assign the key to 110.

To compute the **closeness (or distance)** between a key and a node, giving the scenario in 1.26:

1. Initialize the distance value at 0
2. Compare the key and compute ID bit by bit
3. If the key and the node id differ at the i^{th} least significant bit, add a penalty of 2^i to te distance value. 4 It corresponds to the ***XOR Operation***: it compute the distance betwwen the identifiers and the nodes, interpreted as an *unsigned integer*.

The clostet node is the **Node B** because $4 < C$ and $4 < 9$ and should store **ID, 94.29.160.5, 34665**.

The XOR operation represent a valid **metric for the distance** and guarantees properties like:

- $d(x, x) = 0$
- $d(x, y) > 0$, if $x! = y$
- $\forall x, y : d(x, y) = d(y, x)$ - ***Symmetry***
- $d(x, y) \oplus d(y, z) = d(x, z)$ - ***Transitivity***
- $d(x, y) \oplus d(y, z) \geq d(x, z)$ - ***Triangular inequality***

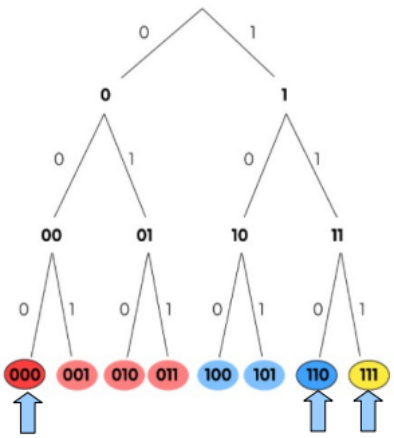


Figure 1.25: Kademlia mapping keys to nodes

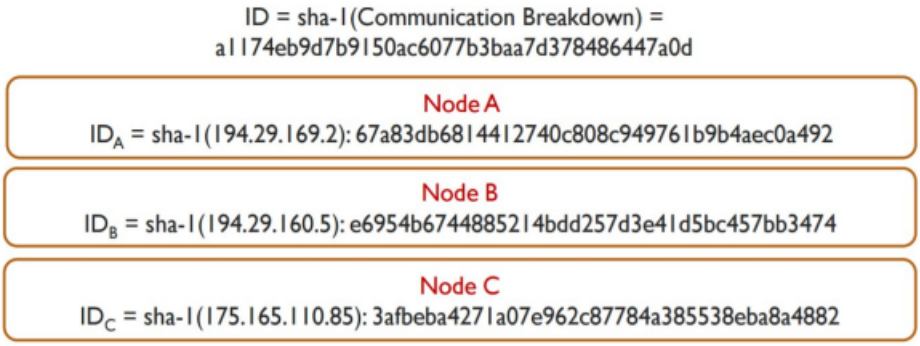


Figure 1.26: Example on ID distance computation

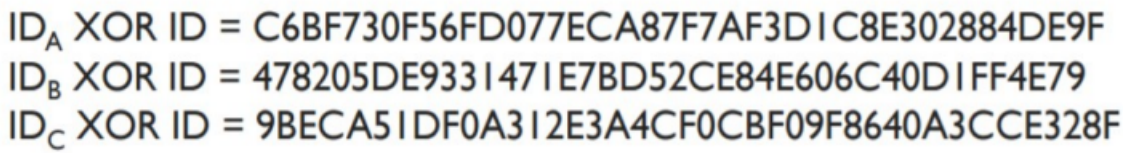


Figure 1.27: undefined undefined

- ***Unidirectionality***: there is a single node at minimal distance with the key. Given x and a distance δ , it exists a single y such that $d(x, y) = \delta$. For example, if $x = 1001, \delta = 0001$ the only point at distance δ from x is $y = 1000$.

The symmetry is showed in a geometric interpretation in figure 1.29: consider 4 peers (*colored one*) the y-axis is the **distance between the colored peers and the peer on the x-axis**.

The distance graph looks the same in both halves but *shifted* along the y-axis: *smaller the distance with any per in the same half space*. The **symmetric** property allows Kademlia to learn contacts from ordinary queries it receives, helping building the *routing tables* (*not symmetric distances like in chords does not allow this*) while **undirectional** property allows to determine a single node ad a *minimal distance* with the key so lookup for the same key **converge to the same path** so caching items along this path is good to avoid hotspots.

Taking two nodes identifiers, **the larger the common prefix, the smallest is the distance (*computed by XOR*)** so close nodes are charetrized by a long common prefix. In figure 1.30 is pictued an example.

As the picture 1.31 clarify, two leaves may be close in the tree and also numerically close, but they are **distant** according to the metrics given by the XOR:

$$1000 \oplus 0111 = 1111 = 15$$

(*which is also the maximum distance in this scenario*) while the *numerical difference* between is 1.

Distances and identifier tree Consider two identifiers x and y of length L that share a **common prefix** of lenght p and differ in the last $i = L - p$ bits, their distance, according to XOR metric will be s.t.:

$$2^{i-1} \leq d(x, y) \leq 2^i$$

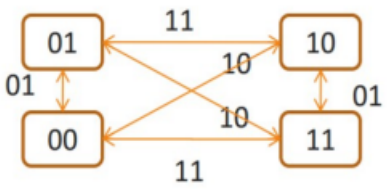


Figure 1.28: undefined undefined

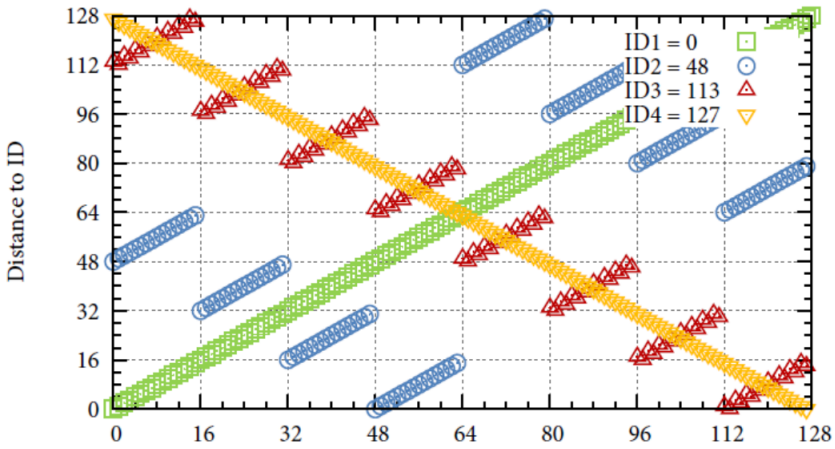


Figure 1.29: Symmetry property under geometric interpretation

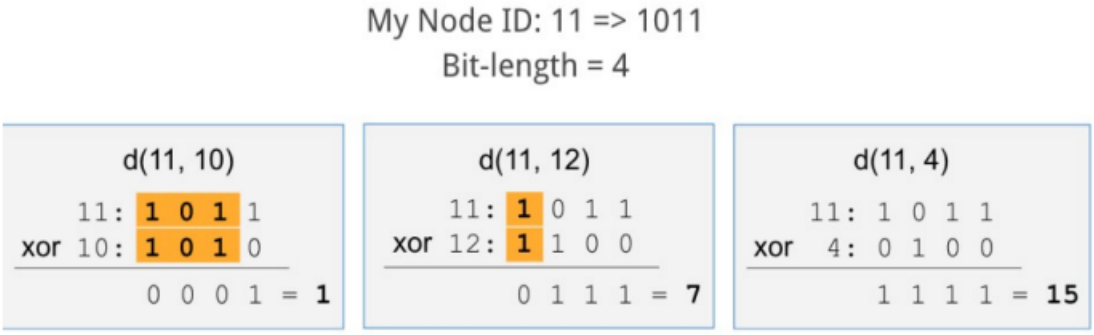


Figure 1.30: Node identifiers distance example

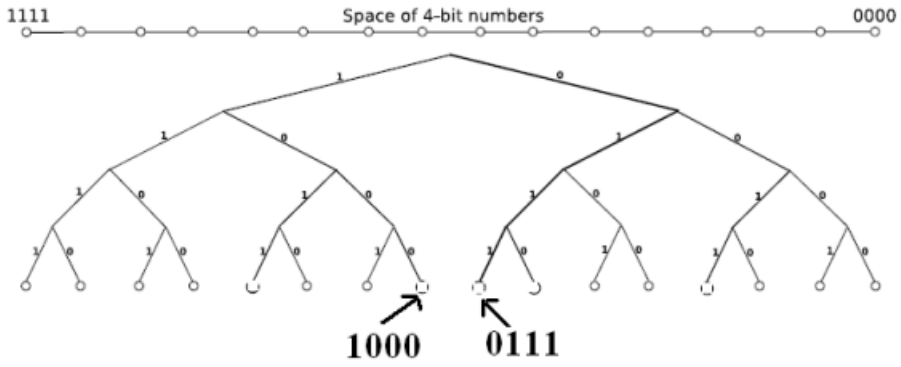


Figure 1.31: Identifier tree

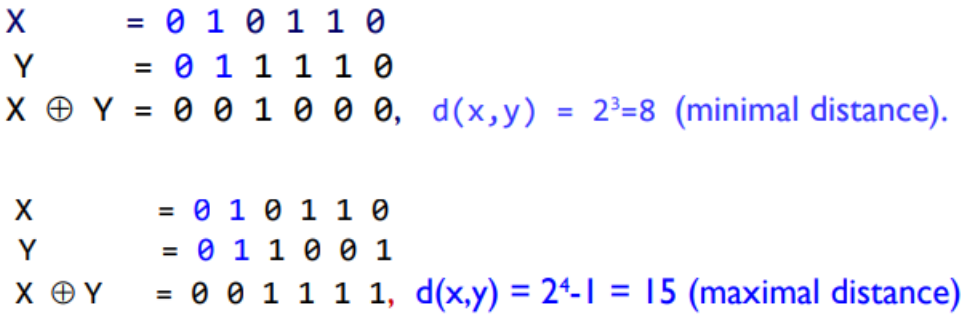


Figure 1.32: `XOR`-based distance computation

An example of *minimal distance* and *maximal distance* are pictured in 1.32

This enables to **pair the nodes** of the subtree with an **identifier range**. Consider the scenario pictured in 1.33, based on the previous one, with a third nodes with identifier `1111`.

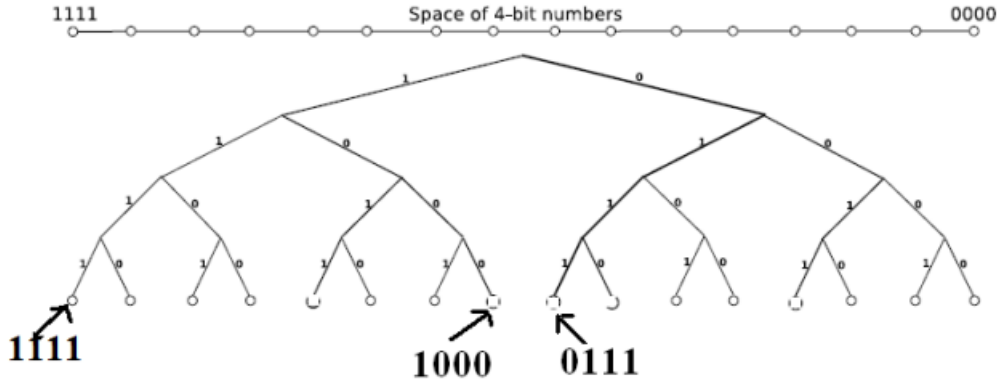


Figure 1.33: Shared-prefix induce a variations of the distance

The leaf 1000 and 0111 have a shared prefix length of 0: the distance varies according to

$$2^3 \leq d \leq 2^4$$

```
0 1 1 1  ⊕  1 0 0 0 = 1 1 1 1 = 15 (maximal, numerical distance is minimal)
0 1 1 1  ⊕  1 1 1 1 = 1 0 0 0 =  8 (minimal, numerical difference is high)
```

Figure 1.34: Refer the identifiers pictured in 1.35

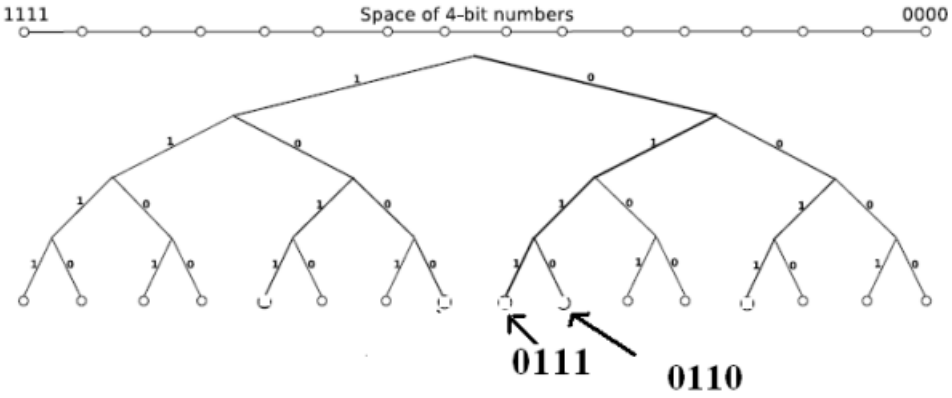


Figure 1.35: Adjacent identifier, refer the computation in 1.34

In figure 1.35, the distance between the two nodes in the figure is *minimal*: they differ only in the last bit so $2^0 \leq d \leq 2^1$ and $0110 \oplus 0111 = 0001 = 1$.

1.4.1 Peers tree: an alternative representation

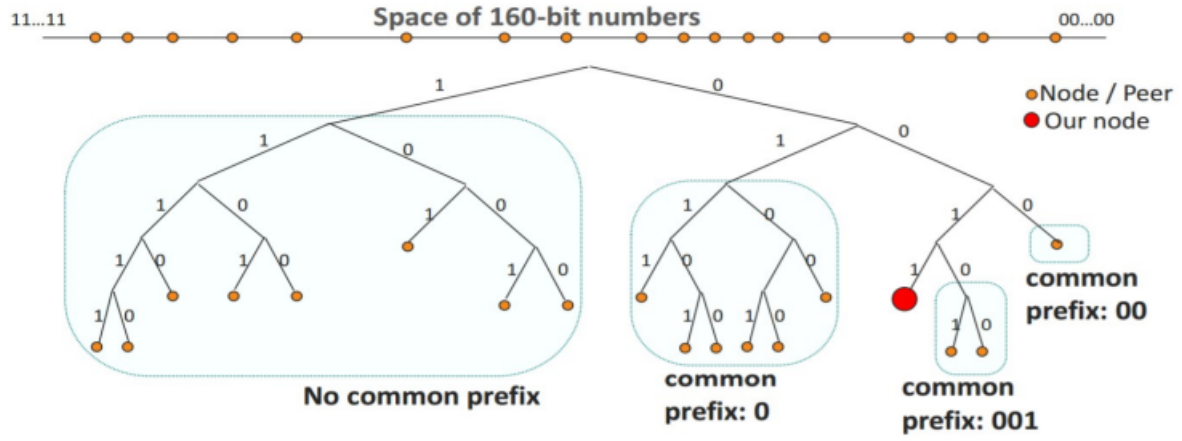


Figure 1.36: Peers alternative representation: a tree

The peers in the network are much lesser than the identifiers because the space of identifier is huge so not all the identifiers are paired with a peer. The *node/peer tree* is an **unbalanced binary tree** showing only the identifiers of peers present in the network (*subset of all identifiers*): **each leaf of the tree is a peer, not for all identifiers**. So a leaf in the node tree corresponds to an identifier prefix: the peer paired with the leaf is the unique node with that prefix so there is a unique correspondence between peers and identifiers (*e.g* 0011 *uniquely identifies the red peer, no other peer have the same prefix and the deepest part of the path is not useful to identify the peer.*)

Routing table

The main goal is to define look-up operation and store only addresses of a small amount of nodes, guarantee a runtime of $O(log(n))$ by storing a log number of node ids and their corresponding IP addresses along some contact taken from the identifier trie. The general idea is sketched in 1.37

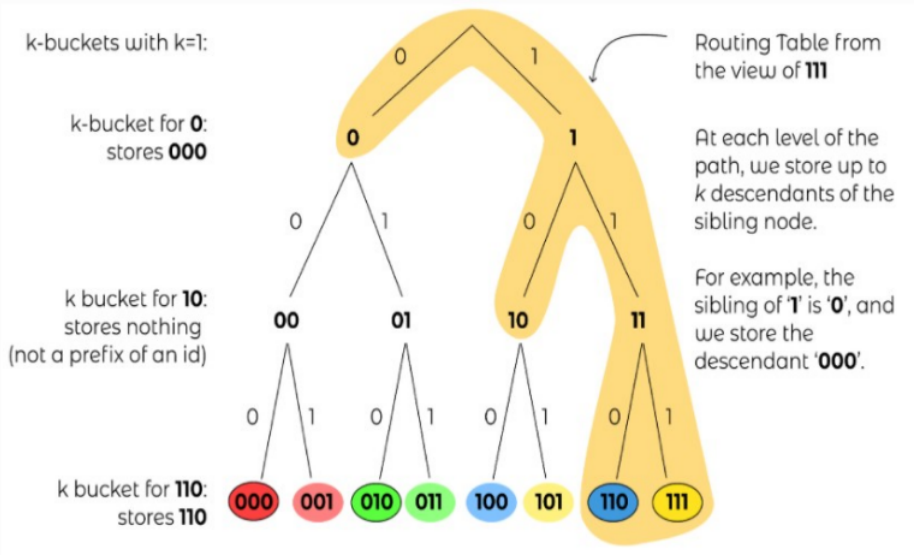


Figure 1.37: Routing table tree view

For a given node, take the the path from that node to the root: for every **sibling node** of a node met on that path

1. store the identifiers of k nodes whose ids are descendent of the sbling
2. create a *sibling bucket* that contains the k contact for each level, forming a **k-bucket**.

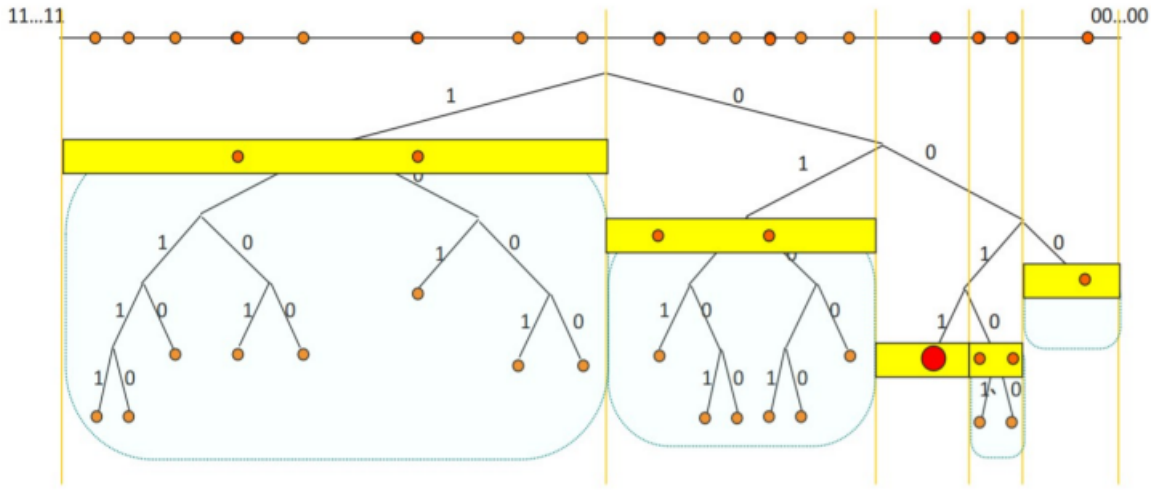


Figure 1.38: Contacts for each read peer, $k = 2$ for each subtree so 4 buckets

The pictured in 1.38 shows the contacts in the routing table of the red peer, including $k = 2$ contact for each subtree, forming 4 different buckets (*in yellow the buckets that contains the contact in the routing table of the* ***red peer***).

The rows of the routing table are **k-buckets*** ($1 \leq i \leq 160$): each row contans k contact and corresponds to a subtree. The contact are stored a *(ID, IP, UDP Port)* and each row contains the contact with distance d from the peer name where $2^{i-1} \leq d \leq 2^i$. So each entry corresponds to a *common prefix*: the lower the entry the longest the common prefix. A general idea is sketched in 1.39.

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

Figure 1.39: Routing table entries

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 entry** of the routing table corresponds to peers sharing a smaller prefix and cover a larget set of identifiers: may include a larger number of contacts but never more than k contacts. The value of k is defined such that the **probability that a crash of more of k nodes is a rare event**. The nodes in each bucket are mantained in ordered such that **least recently contacted nodes are in the first positions of the list**.

K-Buckets management: add contact

As shown in the flow pictured in 1.28, the last peer contacted is putted at the end of the table. If the information does not exists in the buckets, check if the bucket is full: if not remove the least recently node (first position of the list) and if there is no response drop it, applying a sort of *self-healing* update. This approach prefers to mantain the oldest nodes as first choice instead of the new one: this choice is based on analysis made on Gnutella protocol tracing data that shown that *the longer a node has been up, the more likley it's to remain up another hour* so by keeping the oldest live contacts around, k-buckets

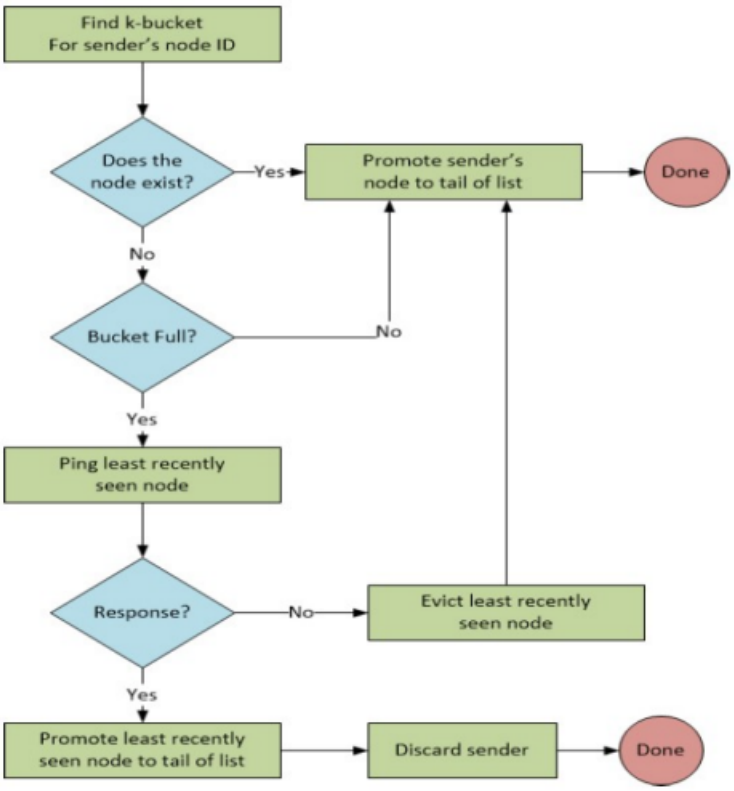


Figure 1.40: Bucket add contact flow

maximize the porbability that the nodes they contain will remain online. This mechanism is also known as **Least Recently Seen Eviction**. Other secondary benefit of the k-buckets is the resistance to the DoS attacks where an attacker cannot flush the nodes routing state by flooding the system with new nodes.

The k-buckets are **refreshed** for each query passing through the node: if a node has left the network, new information received from the queries refreshes the k-bucket list. Can also happen thath a k-bucket is not refeshed for a given period of time due to the lack of messages from nodes in the range covered by the k-bucket. Usually a refresh is **periodically executed by the protocol**: Kademlia chooses an identifier belonging to the range covered by the bucket at random and search the identifier so if the node with that identifier sends a reply it is inserted in the k-bucket.

1.4.2 Key look-up

Given an identifier, retrieve the associate content/node using the XOR metric for minimum distance (*closest node to the key*) as sketched in 1.41

In the previous figure 1.41, is considered $k = 1$ but usually $k \geq 1$. In step 2, pictued in 1.42 the subject node is the **blue node** that contains the reference to the green node, contained in its routing table.

The procedure is *iterative* because the routing information are returned to the original querying node: node n sending the lookup request manages all the search process and at eacj routing step, the node waits for a reply that includes a notification of the next routing step.

To efficiently find the target node, a **parallel routing** is necessary to discover the target node even by routing the requested to the most distant node in the current node routing table. An example is skected in 1.43.

The *blue node* 0011 looks for the red node 1101: it has a reference to the green nodes (1001 and 1110) with:

$$dist(1101, 1001) = 4, \text{ and } dist(1101, 1110) = 3$$

So it's possible that the node 1001, which is more distant from the target, has a reference to the target while the closest node 1110 has no reference to it. The parallel routing allows the blue node to **send**

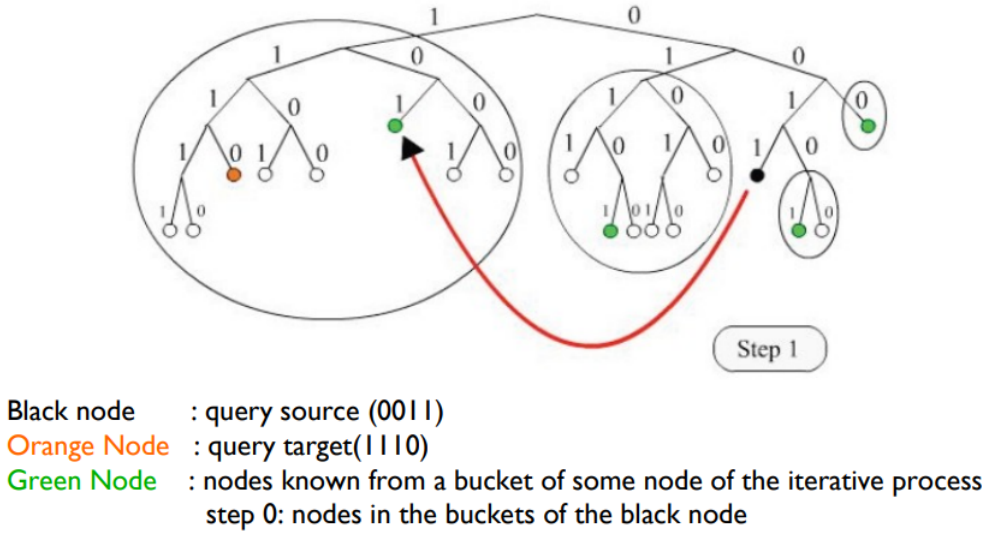


Figure 1.41: Step 1 of key lookup

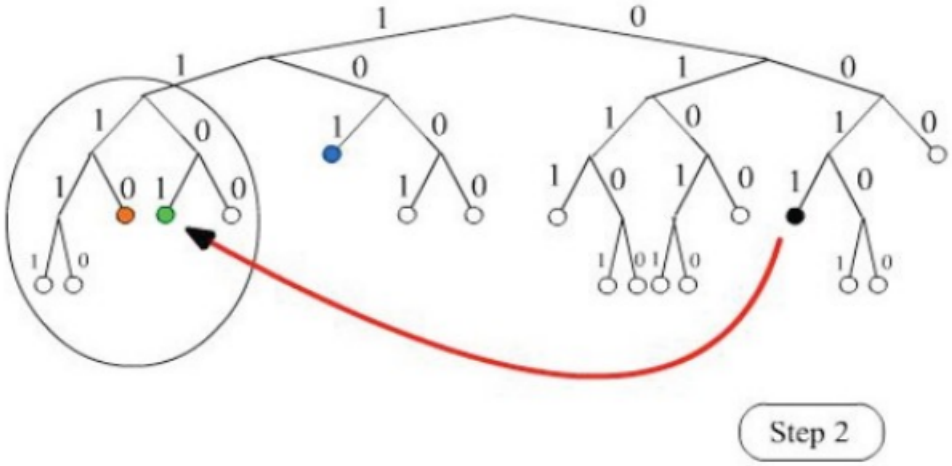


Figure 1.42: Step 2 of key lookup

requests to both nodes. The degree of parallelism is defined by α. The query is sent to $\alpha \geq 1$ nodes closest to target: *the unidirectionality of the XOR metric guarantees that all the pahts converge towards the target.*

1.4.3 Basic Protocol Operations

Kademlia consists of 4 primitives (*not iterative*) operations, defined as *RPCs* that exploits UDP:

1. `FIND_NODE(v) -> w (T)`: `v,w` are nodes while `T` target if the lookup. The recipient of the message `w` return `k` (`IP Add, UDP port, Node ID`) triples for the k nodes it knowns abut the 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.

2. `FIND_VALUE(v) -> w (T)`: the in parameter is a 160 bit ID representing a value. The returned value corresponds to T if is presented in the queried node (`w`) and the associated data is returned. If not present, is equivalent to `FIND_NODE` and `w` return a set of `k` triples. So this operation return a list of other peers, it's up to the requester to continue searching fo the desired value from that list.

3. `PING`: probe node `w` to see if it's online

4. `STORE(v) -> w`: instruct node `w` to store a `<key,value>` pair.

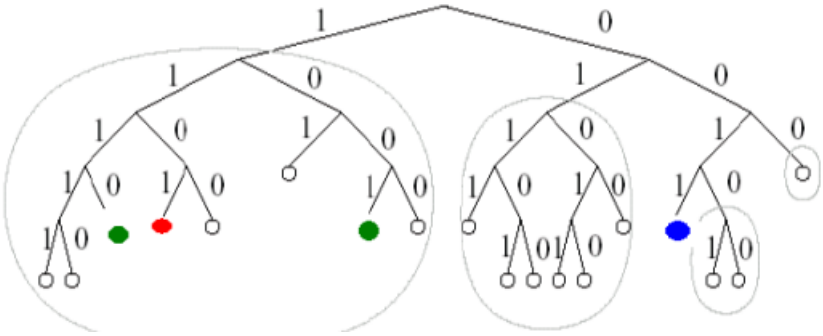


Figure 1.43: Parallel routing for efficient key lookup

1.4.4 Node lookup

To be able to locate the `k` **closest node to a given ID** an iterative algorithm is followed based on the basic primitive `FIND_NODE`: many `FIND_NODE` operations are execute in paralle according to the pamater α that is the sysyem-wide **concurrency parameter**. Based on α value:

- $\alpha = 1$: lookup algorithm is similar to Chord, one step progress each ti,e
- $\alpha \geq 1$: allows the flexibility of choosing any one of k nodes to forward a request to, at each test The *lookup* of a node is exploited both for **finding nodes** and **finding values**: both this type of searching need to stop when the value/node is reached/founded.

Complete lookup example The node `P` looks for the key `Q` (*which is the identifier of a node/content*) (*in figure 1.44*) so `P`:

1. look in the bucket list of the nodes closest to `Q`
2. looks in the k-bucket closest to the key and not empty: if the bucket includes less than α nodes, looks in close buckets.
3. selected contacts may belong to different buckets

   This allows `P` to selects α nodes from the selected bucket: this allows `P` to sends the query in **parallel** to all the selected nodes (*using RPC`FIND_NODE(Q)`*). Each contacted node finds out, in turn, `k` nodes closer to the key: each node may exploit a different bucket of its routing table. The **routing is iterative** because:
4. Each node returns the results to `P`
5. The results are inserted in a list which is ordered on the basis of the distance between the node and `Q`
6. Node `P` continues the routing process through the results obtained from `P`

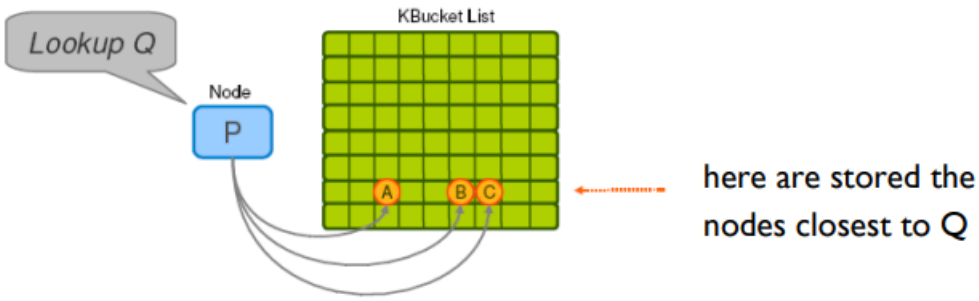


Figure 1.44: Key lookup example: contacts may be in different buckets

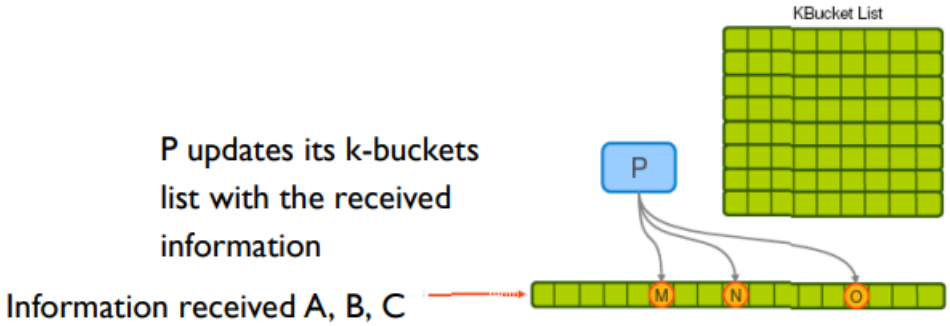


Figure 1.45: Bucket update with received information

At each step, the node `P` selects α nodes from the received information: if it obtains nodes closer to the rarget with respect to the preceding nodes, it performs lookup on these nodes. Otherwise, chooses further nodes from those which have not been still contacted. The algorithm terminate when a round of `FINDW_NODE(T)` fails to return any closer node. The entire algorithm is sketched here:

```
k-closest= alpha contacts from the non-empty k-bucket closest to the key

if there are fewer than a 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 {
1. select from k-closest, alpha closest contacts which have not been queried yet
2. send in parallel, asynchronously FIND_NODE to the delected contacts each contact, if live, returns
3. 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
```

To store a **(key, value)** pair, the lookup operation first find the k closest nodes to the key and sends them `STORE` RPCs. The contacts data is replicated on those nodes. The **republishing mechanism** allows each node to re-publish the (key,value) pair to keep them alive. This mechanism is needed when some of the k nodes that initially get the pair leave the network or when new nodes enter the network with an identifier closer to key than the nodes on which the key-value paiir was originally published. In Kademlia, the **republish** happen every 24 hours.

Node Join/Leave

The `new` node (*or joining node*) borrow an alive node's ID offline (to the *boostrap node* `boot`): the initial routing table of the joined new node has a single k-bucket containing `new` and `boot`.
To enlarge the knowledge of the network, the node need to advertise himself and expand the entries of the routing table: to do this, the node send a `FIND_NODE` with the identifier ID of the node itself. By sending this message, other messages return the reference to the joined neighborhood nodes and other nodes insert the `new` node in their routing table.
Some buckets corresponding to a range of identifiers can be empty: to enlarge the knowledge of those peers the node can also execute a `FIND_NODE` operation with as identifier a subset of the range missing in the routing table.
When the routing table is full in each of k buckets, we can also make an optimization by considering alternative paths by considering the TTL to contact a node: a node can misure the TTL with another one by using the `PING` primitives.

The **leaving operation** is minimal: the node leaving does not require further operations. If a node does not reply, it will be discarted from the k-buckets so by *auto-healing* the topology structucture created by the nodes disconnection.

1.4.5 Strenghts/Weaknesses

Between the weaknesses there are:

1. non-uniform distribution of nodes in ID-space that also result into imbalanced routing table and inefficient routing.
2. the balacing of storage load is not truly solved despite the uses of different path to get the target node: some techniques used in IPFS to solve the storage load balancing use a modification of distribute the content to other peers to be able to manage the load.

Despite, between Kademlia stenghts there are:

- low control message overhead
- tolerance to node failure and leave
- capable of selecting low-latency path for query routing
- provable performance bounds

See a summary here: https://kelseyc18.github.io/kademlia_vis/basics/1/.

Kamdelia as Prefix Matching DHT

Kamdelia is an instance of more general concept of prefix matching due to its routing mechanism based on XOR metrics on prefixes. The basic ideas is a generalization of the routing to hypercurbes: mapping of nodes and keys to the numbers of m digits of a certain base `base`. and assign each key to the node with which it shared the **longest prefix** (*if possible*). Different bases allows to change the pedix of the logarithmic factor in operations. It's the same idea as Kademlia but enlarged to a identification space of base and not only binary. In 1.46 is sketched the basic idea:

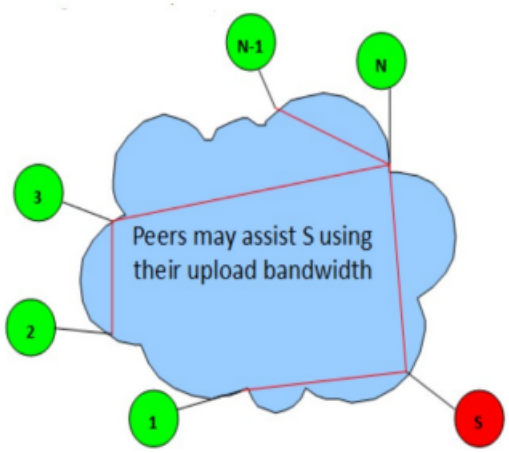


Figure 1.46: Prefix matching DHT

The initial file request are served by a centralized server: further requests served by peers which have already received an replicated the files.

1.4.6 Content Distribution Network (CDN)

A **Content Distribution Network (CDN)** is a distributed network of servers that are designed to deliver web content to end-users with high availability and performance. The CDN works by caching web content, such as images, videos, and static HTML pages, on servers located around the world. When a user requests a particular piece of content, the CDN will route the request to the nearest server that has a cached copy of the content, reducing latency and improving response time.

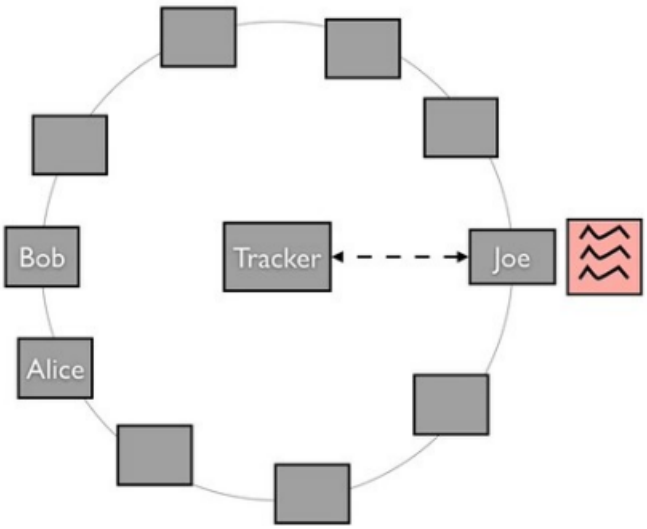


Figure 1.47: CDN example: interaction between clients and tracker

CDNs typically consist of three main components: **edge servers, a caching infrastructure, and a global network**. Edge servers are distributed geographically to bring the content as close to the end-users as possible. The caching infrastructure is used to store and manage the content, and the global network is responsible for routing traffic to the appropriate edge server.

As an example, let's consider a popular e-commerce website that sells products globally. The website's static content, such as images, product descriptions, and videos, can be cached on a CDN. When a user requests a product page, the CDN will route the request to the nearest edge server that has a cached copy of the page. This reduces the latency and improves the user experience. The CDN can also provide security features like DDoS protection and SSL/TLS encryption to improve website security.

Some popular CDN providers include Amazon CloudFront, Akamai, Cloudflare, and Fastly. These providers offer a wide range of features and configurations to meet the needs of various websites and applications.

1.5 BitTorrent Protocol

BitTorrent is a **content distribution network**, not based entirely on P2P paradigm. The problem is how distribute the contents to hundres of thousends of simultaneous users. The *efficient content distribution* is developed using **file swarming**, later discussed, which is based on the idea that *a peer makes whatever portion of the file that is downloaded immediately available for sharing.* Some issues have to been addressed:

- Flash crowd phenomena: large number of request in small time period implies the ability to manage a high degree of unexpected traffic
- CDN (Content Delivery Network): data and services replication on different mirror servers to optimize bandwidth usage

The basic idea is to import the CDN mechanism in a P2P environment, without the intervention of a centralized server: differently from a P2P system, it does not perform all the functionsof a typical P2P system like *searching.*
The initial file reques are served by a centralized server but further requests served by peers which have already received and replicated the files. The server distribute two further pieces: in parallel, so previous pieces are distributed. When all pieces are distributed among peers, the server is no more inolved in the file dsitribution. The distribution and recomposition on request of those piaces of file allows to avoid an unexpected and high load on a single node/server.

In summary, more nodes can serve thc ontent, not only one server: there is the need to detect which node is currently providing the content by introducing the **tracker** that taking trace of who is currently providing the content. In the example, *JOE* connects to the tracker announcing the content he store and the tracker update the information *"JOE provide the content x"*. If *BOB* want to download the red content:

1. Open a connection ti the tracker and learn that *JOE* is providing the content
2. Establish a direct connection with *JOE*
3. *JOE* sends the content, from now on the distributon of the file goes on the directly between peers
4. *BOB* then announces the tracker that is can also provide the red content, as pictured in 1.48.

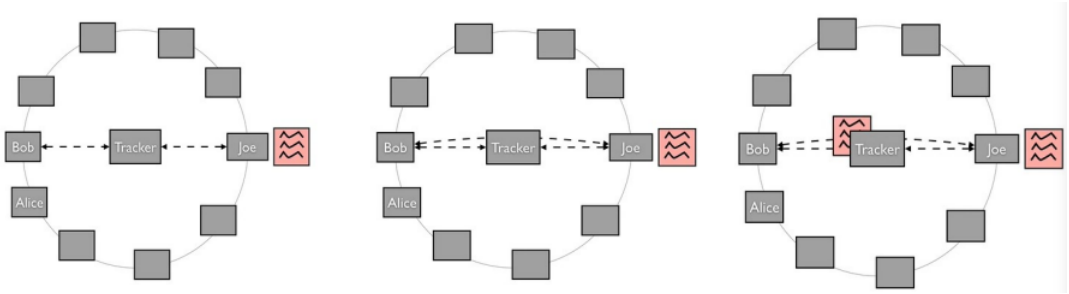


Figure 1.48

Glossary

A general summary is sketched in 1.49, with the following elements:

- **Descriptor**: The file descriptor `.torrent` are managed by centralized server and not fully by the nodes. The descriptor includes the reference to a *tracker*.
- **Tracker**: entity which coordinates the peers sharing the file
- **Swarm**: set of peers collaborating to the distribution of the same file coordinate by the same tracker
- **Seeder**: peer which own all the parts of the file, having reconsturcted the entire content. At starting time a single seeder S exists
- **Leecher**: is a peer which has some part of no part of the file and downloads the file from the seeders and/or from other leechers.

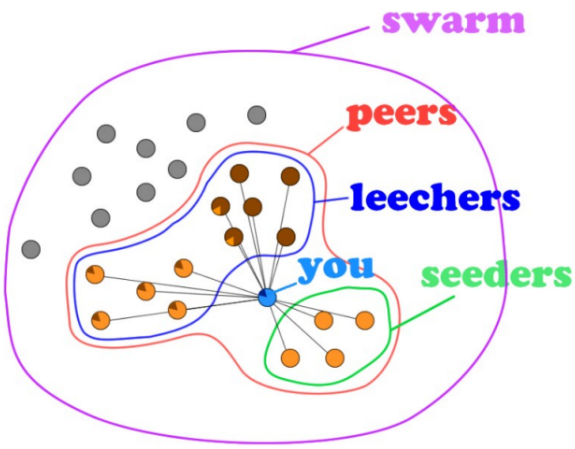


Figure 1.49: BitTorrent key elements

1.5.1 Protocol overview

The protocol is briefly sketched in 1.50.

In the pictued schema, the **peer Seeder** is going to share a file by:

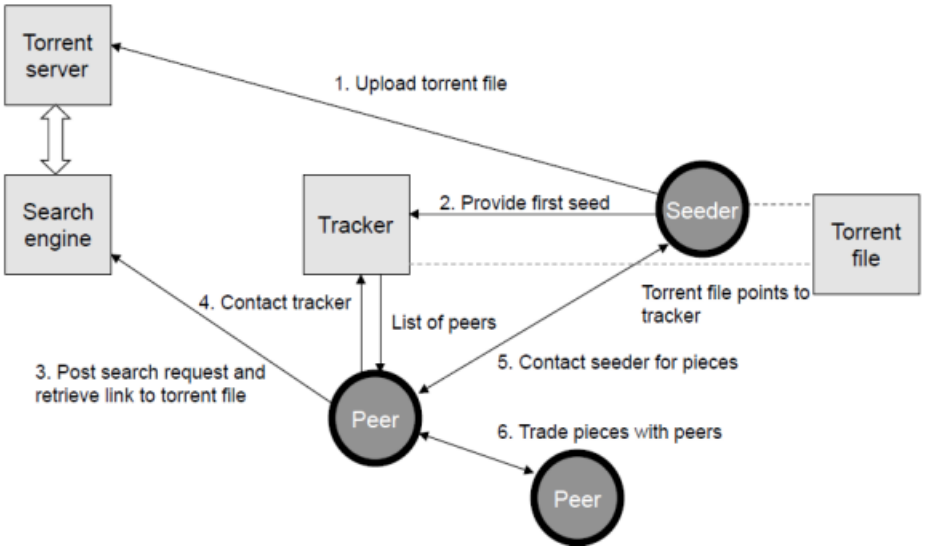


Figure 1.50: BitTorrent protocol interaction

1. Upload the `.torrent` on a Torrent Server
2. Opens a connection to the **tracker** and informs it of iwn own existenceUnknown Node :: textDirective the moment, it's the only peer which own the file The **peer Peer** is going to download the file by:
3. Retrieve the *file description (`.torrent`)* and opens it through the BitTorrent client
4. Open a conenction to the tracker and informs it of itw own existence and recieve from the tracker a list of **peers of the swarm**
5. (and step 6): opens a set of connections with other peers of the **Swarm** so ask other peer about the parts they own, declaring its interest in some part of the file. Continues by exchaing the information with the peers in the swarm. If **Peer** remains online when it has finished the file download, it goes on distributing the file, so becoming a **Seeder**.

In picture 1.51 is summarized the explained protocol.

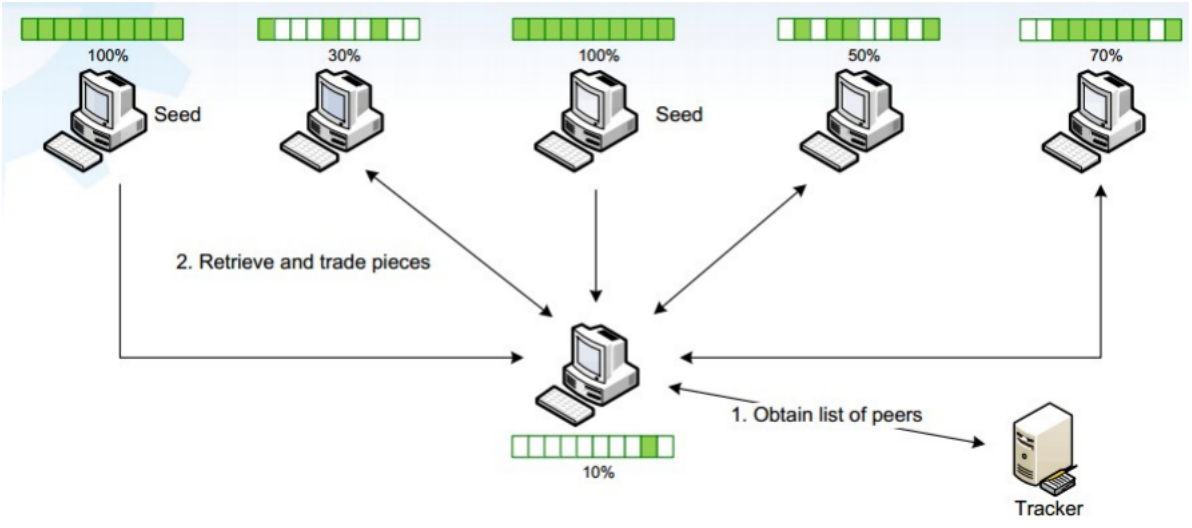


Figure 1.51: undefined undefined

Pieces and suboieces (blocks)

The content is split into chucks called **pieces** of a choosen dimension between $256KB$ and $2MB$: later those pieces are divided into **blocks** of $16KB$. The hash of **each piece** in $SHA-1$ is stored in the `.torrent` file descriptor: pieces are the *smaller unit of retransmission* so when the pieces are retrieved from seeders, the hash in file descriptor and the computed one are compared to verify the correctness of the piece to determine if retransmission is needed. (1.52)

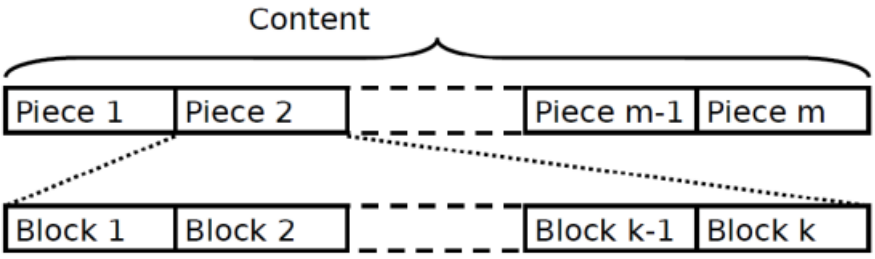


Figure 1.52: Formation of pieces and blocks of BitTorrent content file

1.5.2 Publishing

Torrent publishing is a process of distributing large files over the internet using a peer-to-peer (P2P) network. The process involves breaking the file into small pieces and distributing them to multiple users or peers. Each peer downloads and shares small pieces of the file with other peers, allowing for faster and more efficient downloading.

To publish a file on a torrent network, the following actions are typically taken:

1. Create a Torrent File: The first step is to create a small metadata file known as a torrent file (1.53). The torrent file contains information about the file being distributed, such as its name, size, and checksums of each piece. It also contains a list of tracker servers that are used to coordinate the sharing of pieces between peers.

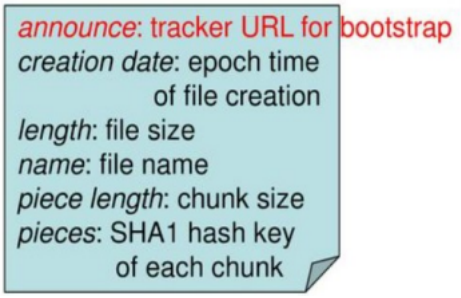


Figure 1.53: Torrent file metadata

2. Upload the Torrent File: The torrent file is then uploaded to a torrent website or tracker server. The tracker server acts as a central coordinator between peers, keeping track of which peers have which pieces of the file.

3. Share the Torrent File: Once the torrent file is uploaded, it can be shared with others through various means such as email, social media, or file sharing platforms. Peers can then download the torrent file and open it using a torrent client.

4. Download the File: After opening the torrent file with a torrent client, the client connects to the tracker server and begins downloading the file by requesting pieces from other peers. As pieces are downloaded, the torrent client verifies their integrity using the checksums in the torrent file.

5. Seed the File: Once the file has been fully downloaded, the torrent client can continue to upload or "seed" the file to other peers. This helps to improve the download speeds of other peers and contributes to the overall health of the torrent network.

The `.torrent` file descriptor are automatically generated by ad-hoc tools like *maketorrent* or by the BItTorrent client: some informations are required by the client like set-up info, address of a known tracker etc. To serialize the content it's used the **Bencode**: it's a serialization code similar to JSON but only used in `.torrent` files. The bencode serialization provide only 4 datatypes: `String, Integer, List, Dictionaries` and a set of simple rules, here listed:

1. Strings: A string in Bencode is a sequence of bytes. It is encoded as follows: `<length>:<string>`. Here, `<length>` is the length of the string in bytes, and `<string>` is the actual string.

For example, the string "hello" would be encoded as `5:hello`.

1. Integers: An integer in Bencode is a number encoded as follows: `i<integer>e`. Here, `<integer>` is the actual integer value.

For example, the integer 42 would be encoded as `i42e`.

1. Lists: A list in Bencode is a collection of other Bencode data types, including strings, integers, lists, and dictionaries. It is encoded as follows: `l<item1><item2>...<itemN>e`. Here, `<item1>...<itemN>` are the individual items in the list.

For example, a list containing the integers 1, 2, and 3 would be encoded as `li1ei2ei3ee`.

1. Dictionaries: A dictionary in Bencode is a collection of key-value pairs, where the keys are strings and the values can be any Bencode data type.
It is encoded as follows: `d<key1><value1>...<keyN><valueN>`. Here, `<key1>...<keyN>` are the keys in the dictionary, and `<value1>...<valueN>` are their corresponding values.

For example, a dictionary containing the key "name" with the value "Alice" would be encoded as `d4:name5:Alicee`.

Peer Bootstrap

In the boostrapping phase, a peer downloads the `.torrent` of the file it wants to download from the tracker: the peer retrieve the tracker's `URL` and connects to it by issuing an `HTTP GET` request. Then sends to the tracker the informationa bout its identity (*identifier, port, etc*), **the number of the peers of the swarm it needs** and other minor informations.

The **tracker** return a *random list* of **50 peers** already in the torrent, providing information for each peer like *peer ID, peer IP, peer Port*. **Tracker** are used for boostrap and as a **certification authority** but are not inolved in file distribution itself: if a `.torrent` file is certified by the server, there is no way to corrupt the torrent because each $SHA-1$ in the `.torrent` is used from peers to verify the integrity of the pieces received from the P2P Network. A peer connects to some peers returned by the tracker open at most *40* outgoing connections so remaining peers keps as a pool of peers to connect if needed.

A node uses only part of its bandwidth to boostrapping while another part is left free to accept incoming connections.

Why blocks?: Pipelining The idea to divide **pieces** into **blocks** derive from the fact that the BitTorrent protocol works on TCP Protocol so the underlying **Slow Start Mechanism** of TCP enter in action: if data are not regularly sent, TCP brings the peers connection to a slower speed than normal. It's crucial to have always something to transfer on a given connection to sustain the transfer rate constant. Each peer for each connection have always some blocks (*typically 5 blocks*) ready to sent for each connection.

The mechanism that allows to determine the order in which the pieces are selected is called **Peer Selection**: this mechanism is fundamental because an inefficient policy may end up in a situation where each peer has all identical set of easily avaiable pieces and none of the missing ones. Several policies that provide efficiency exists:

1. Strict Priority if a subpiece of a piece has been required, all the other subpieces of the same piece are required before requiring another block, as sketched in 1.54. It favors a fast assemblage of the pieces: only the complete pieces may be exchanged with other peers so each peer tries to assemble in the fastest way complete pieces.

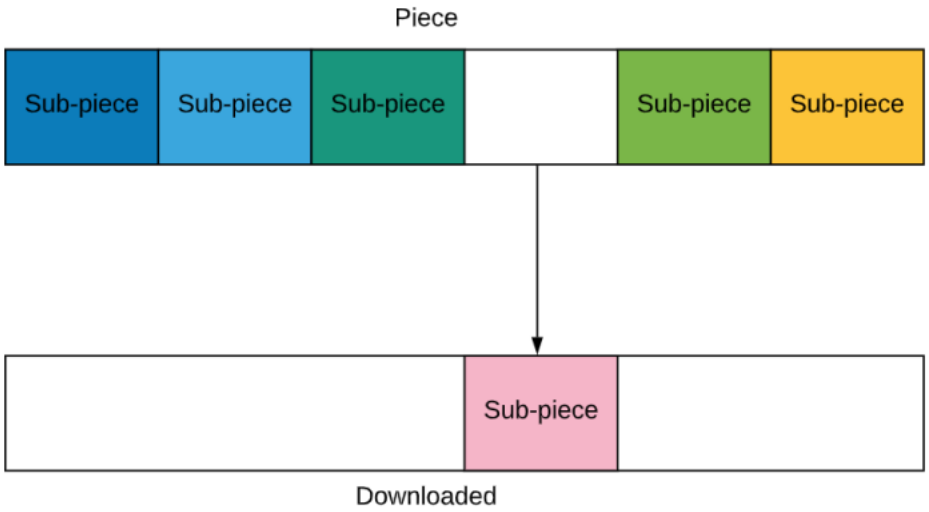


Figure 1.54: Strict priority order enforced by Peer Selection

2. Rarest First Each peer knows the pieces owned by the others so it have a **local view** (*by gossip algorithms*) of the **availability of each piece**: this allows to form the **Rarest Piece Set** that contains the pieces with the *minimal availability*. The algorithm select a piece among the rarest, as pictured in 1.55 and 1.56:

Imagine that there are a set of peers, the edges indicates the ownership of a piace: the pink piece is the rarest. So the upload rate is higher of the seed and because the pink piece is highly requested, in exchange everyone want to download from us so we can download faster from others. It's likely that the peers that own the rarest piace will be insweted in the upload list of several peers.

One **advantage** is that it can increase download speeds because the more peers that hold a piece of the file, the faster the download. P2P networks can also prevent rarest piece missing, which can make it impossible to assemble the entire file if a seeder disconnects from the swarm. Additionally, using a P2P network can decrease pressure on the seeder because new downloaders can download parts of the file from other peers. This guarantees that the entire file is replicated and the download load on the seeder is decreased.

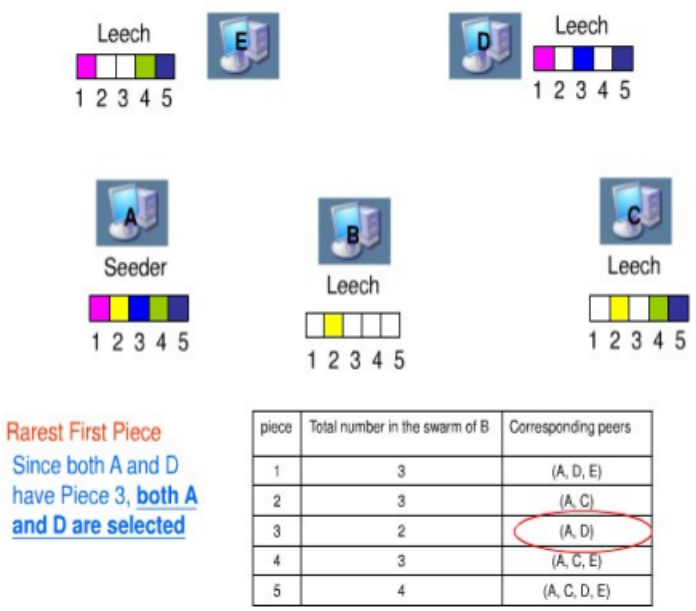


Figure 1.55: Rarest selection process among nodes

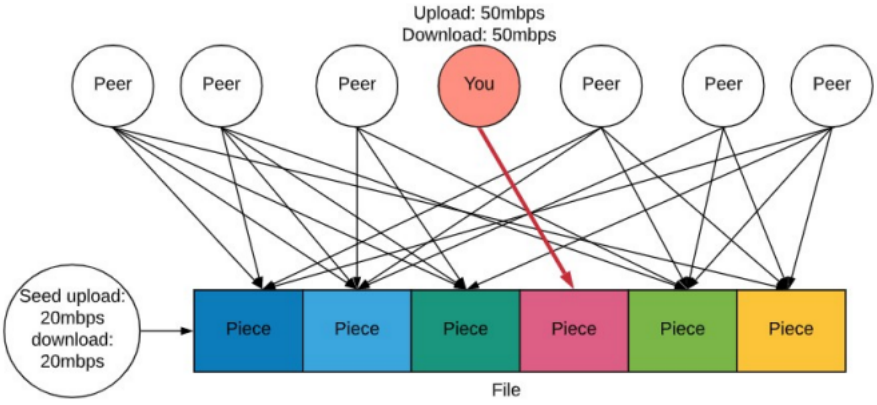


Figure 1.56: Rarest pattern behavior

3. Random First Piece Initially, a lecher does not own any piece and cannot offer anything to the other peers of the swarm: it is important that a **peer acquires a piece as soon as possible, to start the negotiation** of the pieces with other peers (***unchoking algorithm***). In the *rarest first* politics is not initially exploited because the download of a rare peer may be slow the process, because:

1. less piece availability, larger request
2. larger probability to choose a slow peer
3. sharing of upload bandwidth of the owner of the rare piece

This introduces the *Random First Piece Policy* at least for the **first 4 pieces to download**: they are chosen at random. This allows a **fast starting** and, after that, the *rarest first policy* is exploited.

4. Endgame The last piece is downloaded using the **Endgame** policy: in the final phase the peers are characterized by low bandwidth, slow down when transfer is going to end.

The key idea of this policy is that the last pieces of the file to download are required to **all the peers that own the file**: to avoid bandwidth waste, when the piece is received from the peer which has requested it, the download started in parallel are cancelled. There is only a little waste of bandwidth, since the end game is executed for a small period of time, as showed in 1.57.

1.5.3 Free riders problem

The *free riders* in a general definition are "*individuals hiding his/her preference for a common good, so avoiding to pay its price and giving to others the burden to pay the good*". So if there are other

Figure 1.57: undefined undefined

individuals interested in the common good, the free rider is aware that he/she can benefit of the common good without paying for it.

In BitTorrent, the *free riders* are **peers that do not put their bandwidth at disposal of the community**. The solution to this problem in P2P context is complex due to an absence of centralized entity that may control the nodes and it's not possible to impose a given behaviour to the BitTorrent clients, because it's possible to modify the of the client by reverse engineering. The used approach is to based on a *dynamic monitoring of the connection*: the **reprocity approach** states that *a client obtains a good service if and only if it gives a good service to the community*, implemented by ***Choking/Unchoking Algorithm***.

Choking Algorithm

Each peer, periodially, *evaluate its neighbour* 's download speed in the previous round and decides which neighbours it's going to *"choke"*. The decision on which peers to *unchock* is taken every 10 seconds by each peer ans it's based on:

1. The download rate of the last X seconds Because the use of TCP slow-start mechanism, rapidly choking and unchocking is bas thus the timeframe of 10 second is suitable to match the timeframe of the slow-start mechanism. If our upload rate is high more peers will allow us to download from them. This means that **we can get a higher download rate if we are a good uploader. This is the most important feature of the BitTorrent protocol**.

In 1.58 is a pictured scenario:

Figure 1.58: Leecher exchange

BitTorrent divides the time in **round** (*generally 10 seconds*): for each round, decides **to whom it wants to send data (upload)**. Each connection with a peer in the neighbor set is controlled by the following variables:

- `interested/uninterested`: want/do not want a piece from you? (download)
- `chocked/unchocked`: want/do not want to send data to you? (upload) The connections are **bidirectional** so each connection has associated a pair of the previous listed variables.

Each peer maintains for each remote peer it is connected the following state:

- ***am_choking***: the local peer is choking the remote peer. it does not want to send data to the remote peer because the local peer is not satisfied about how this peer collaborates
- ***am_interested***: the local peer is interested in at least one piece on the remote peer
- ***peer_choking***: the remote peer is choking the local peer. The remote peer does not want to send data to the local peer, because it is not satisfied by the collaboration with the local peer
- ***peer_interested***: the remote peer is interested in at least one piece of the local peer

So the *local peer* can receive data from the *remoe peer* if:

1. the local peer is **interested** in the remote peer, and
2. the remote peer **unchoked** the local peer, as showed in 1.59

Optimistic Unchoking

The previous algorithm does not take into account the newly entered peers because initially have nothing to trade and not obtain any interesting pieces. To overcome this limitation, BItTorrent adopt **optimistic unchoking**: one random selected peer is unchocked so the current download rate from that peer is ignoreted and allow newcomers to download resources.

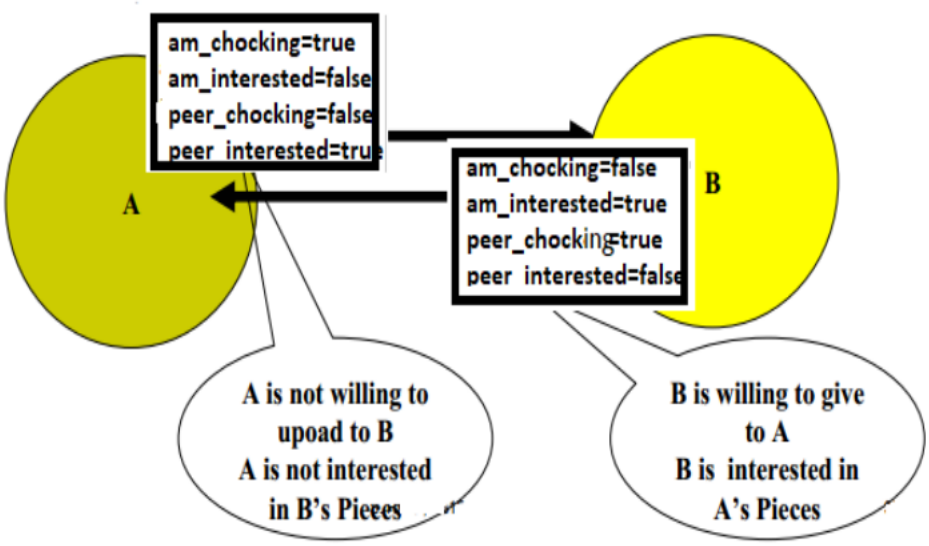


Figure 1.59: Choking scenario between node A and B

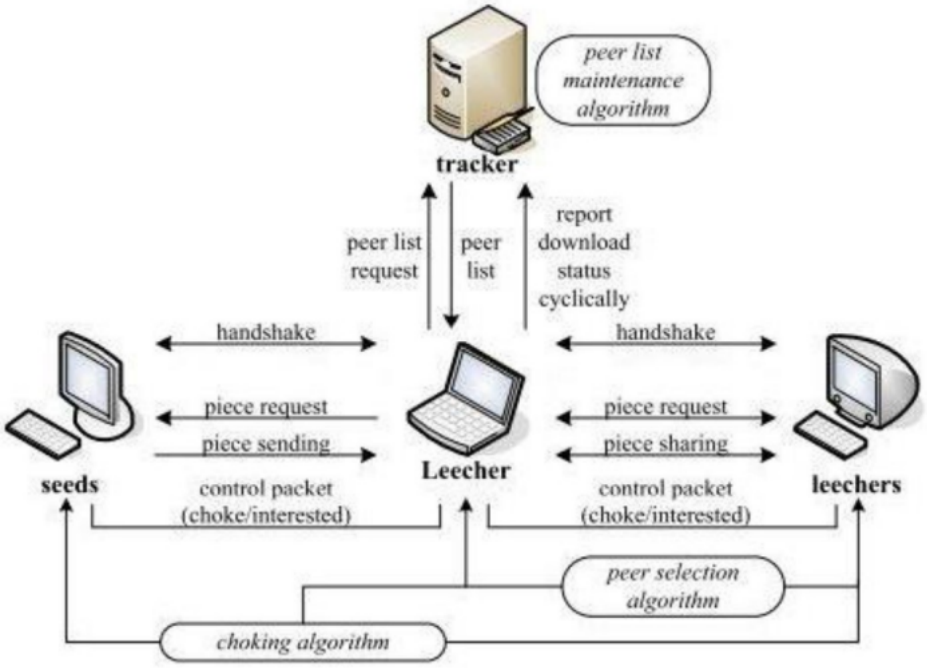


Figure 1.60: BitTorrent at a glance

1.5.4 BitTorrent at a glance

The overall exchange specified by the protocol is pictured in 1.60.

Horizontal arrows represents the exchange of messages in P2P fashion (as discussed till now).

Two-way handshake

Once a TCP connection is established with one of the peer received by the tracker, a client perform a two-way handshake. The header of the message 1.61 contains:

- name length:"0x13"
- protocol name: "BitTorrent protocol"
- "Reserved": used to signal extensions to the basic protocol.

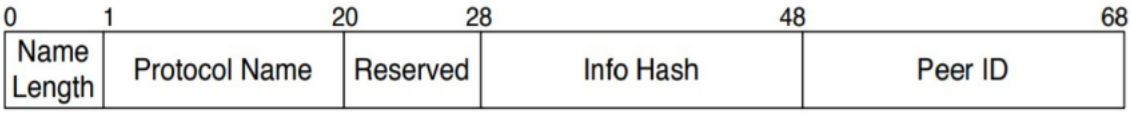


Figure 1.61: Header of BitTorrent message

- SHA-1: hash of the torrent file's infohash that contains the hash of the content the peer is looking for.
- peerID: the client random ID

The peer which receives a TCP connection request **may refuse the connection** based on two checks:

- if the hash does not correspond to **any hash of the files it is participating to download**. The tracker has returned the hash of each pieces so if the hash is not present, the connection is not allowed.
- if the ID of the peer does not correspond to those received by the tracker so the ID is not present in the *swarm* group (*returned by the tracker.*)(1.62)

Peer Wire Protocol

Then, after the handshake, the first thing that peer exchange is the `<bitfield>`: it's a bitmask in which if bit i is set to 1 the peer has the piece i, 0 otherwise. The exchange is sumamrized in 1.63

The *message exchanged* are:

- **Interested**: sent from a peer A to another peer B. It indicates that A is interested in any of the B's pieces and contains the piece index.
- **Unchoke /(Choke)**: sent from B to A if B unchokes A. It notifies that the request of download coming from A is accepted.
- `Request<index, begin, length>`: this message is sent from peer A to peer B to *request a subpiece (block)* with `<index>` index and starting with an offset `<begin>` within the piece, of length `<length>`. A requested can be sent **only after receiving Unchoke message from B**.
- `Piece<index><begin><block>`: only one message is used to *send pieces*. The message is sent from peer A to B to send a block to peer B: the block of index `<index>`, starting with offset begin within the piece and payload `<block>`.
- **Have**: notifies that a new piece is completely downloaded and is available for sharing. It contains information useful for monitoring the swarm: who has what. When a peer receives a Have message, it updates its view of the swarm.
- **Not Interested** indicates that the sender is not interested in that piece anymore. May be sent after receiving a HAVE message
- **Cancel**: sent from a peer to another to indicate that it already got a piece and it is not interested in it anymore. It's sed in end game mode only: sent from peer A to peer B to cancel a request already sent to peer; for the piece with index starting with an offset within the piece of a length.

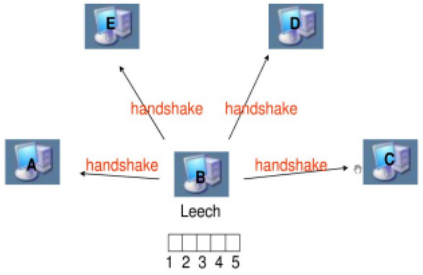


Figure 1.62: Nodes handshake with the leecher

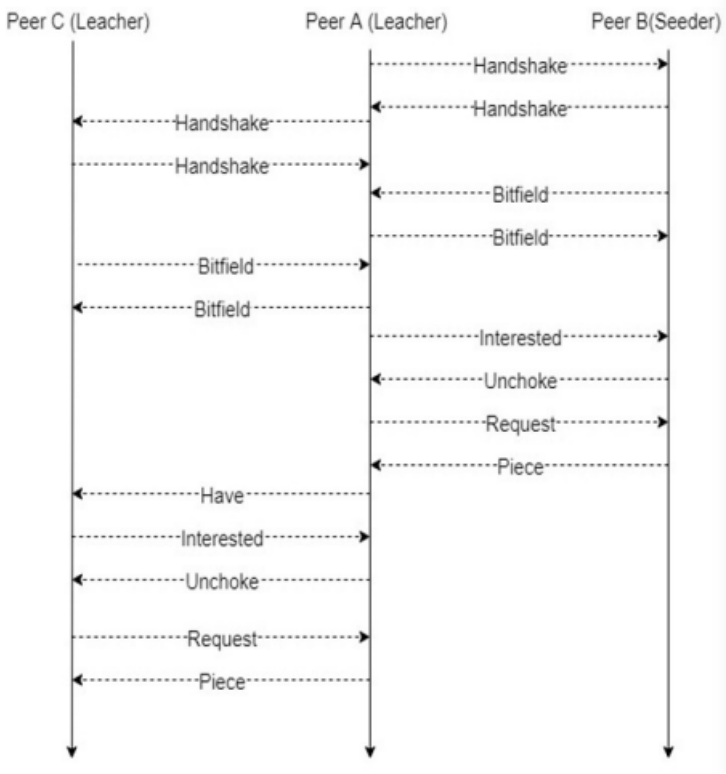


Figure 1.63: Peer wire protocol exchange

1.5.5 BitTorrent Mainline DHT

Bittorrent Mainline DHT allows to **decentralizes the tracker services** that usually represent the bottleneck. This allows to o provide a "trackerless" peer discovery mechanism to locate peers belonging to a swarm by asking to the DHT, instead of the tracker. Each node of the DHT **stores a part of the tracker information**: now each peer implement two different functionalities:

1. a client/server listening on a TCP port that implements the BitTorrent wire protocol
2. a client/server listening on a UDP port implementing the distributed hash table protocol. So the DHT plays the role of the tracker and **stores the content's infohash (*key*)** and **the list of the peers in the swarm (*value*)**.

The protocol messages includes basic messages like:

- `PING`: probe a node's availability or announce one's existence
- `GET_PEERS(H)`: look for peers belonging to the swarm for the content with InfoHash H. The contacted node may
  - store peers for H, replies with a PEER message containing them
  - have no info for H: replies with the identifiers of the 8 peers closest to H
- `ANNOUNCE_PEER`: peer announces it belongs to a swarm.
- `FIND_PEER (target_id)`: request for nodes which are close to the node with node ID target_id

Example

For example Alice wants to download the piece of content stored by Bob and identified by $id = 14$. Alice only knows the peers $6, 7, 13$. Following the Chord idea of contacting the clostest node (*with XOR distance to the content as in Kademlia*), the clostest to the result node is peer 13. Now 13 does not known who has piece 14, so it sends back to Alice the clostest peer to it, being peer 15. Peer 15 knows that peer 4 has the content, and sends the contact information to Alice, which can then use the BitTorrent TCP protocol to exchange the content with peer 4.

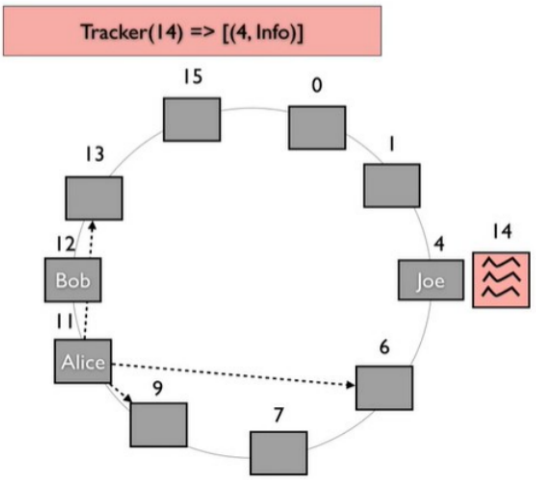


Figure 1.64: Node contacted by Alice

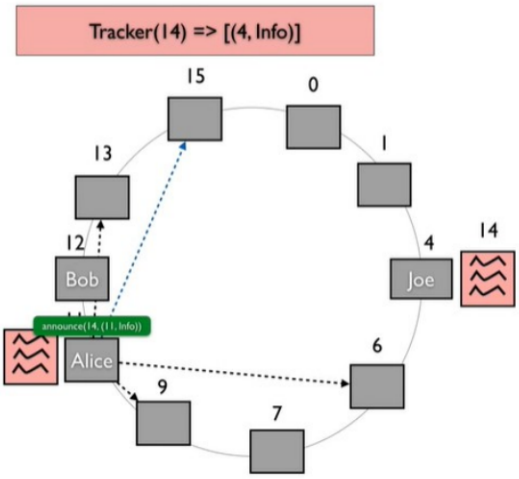


Figure 1.65: Broadcast message

After downloading the piece, Alice sends (*in broadcasts*) an **announce** message to tell all the peers it stores piece 14 and can exchange it, as shown in 1.65.

The announce message has key 14 and value $(11(peerid), info = (ipaddr, port))$. All the other nodes will add it to their "tracker"informations. The internet uses NAT to face the scarcity of IP address, this is a problem for Peer to Peer network.

Chapter 2

Cryptographic tools and data structure

2.1 Cryptographic toolbox for DHT and Blockchains

The following tools are used for the development of DHT and blockchains: they include hash function, cryptographic hash functions and digital signatures. More advanced tools are accumulators and zero-knowledge principles.

2.1.1 Hash function

It's a function that converts a binary string of arbitrary length to a binary string of fixed length. The **input** is described by its length counted in bits, normally a maximum length but even zero length is permitted. The **output** represent the compression and usually have a fixed length among $128 - 160 - 256 - 512$ bit.

There are noticeable differences among **non-cryptographic** and **cryptographic** one hashing function: in case of 8-bit block parity (*as shown in picture 2.1*), it's easy to find a collision by inverting any even number of bits in m that are in the same column and the parity will not change.

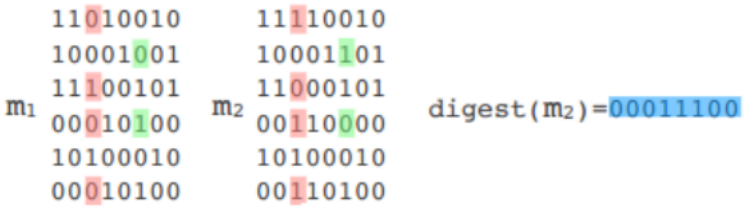


Figure 2.1: Collision by inverting even number of bits

One of the main property desiderable from cryptographic hash function is the minimum probability to find a value that determine a collision. Another example is the **CRC - CyclIc Redundancy Check**: the CRC is the remainder in a long division calculation. In the past has been used to detect burst errors. The **hash collision** are inevitable since the codomain of the hasing function is smaller than the domain. It follows the **Pigeon Principle** for which if n items are put into m containers, with $n \geq m$ then at least one container must contain more than one item. The **hash security** implies that it's very hard to *find collisions* because would require a huge amount of computational power. To bound the probability of having a collision, it's necessary to determine the maximum number of guesses to certainly find a collision by a brute force attack.

Using a brute force approach, picking $2^{256} + 1$ distinct values in the domain and compute the hashes of each of them, hecking if any two output are equal: for the pigeon principle at lest a collision will be founded. The maximum number of guesses required is:

- $O(2^n)$ time complexity
- $O(1)$ space complexity
- $n = len(H)$ number of elements to try

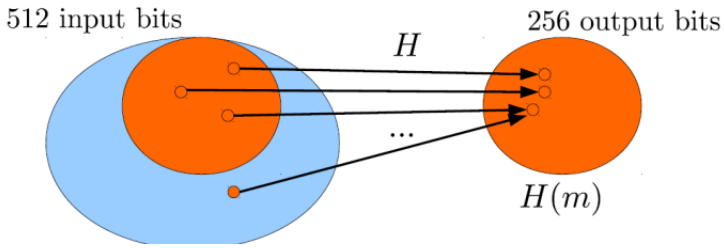


Figure 2.2: Hash function domain and codomain set

The **birthday paradox** show the correlation between the size of the domain and the maximum number of samples required. Specific for the paradox, it states that in a room with n people, the value of n to have the probability that two share a birthday becomes larger than 50% under the hypothesis of.

- a year of 365 days
- all days equally probable This result in the correlation of $n = 23$ ($\sqrt{(365)}$)

The paradox show a connection with hashing function H with n possible outputs: there are 2^n possible different hashes. If H is applied to k random inputs, what must be the value of k so that the prob. that at least a pair of x, y satisfy $H(y) = H(x) = 0.5$? If hashing about $2^{n/2}$ random values then you expect to find a collision because $\sqrt{(2^n)} = 2^{n/2}$.

2.1.2 Cryptographic hash functions

A *cryptographic* hash function is an hash function that minimize the **chance of collision** if the inputs are chosen at random: in case of an adversarial that look specifically to create a collision, it's necessary to guarantee a more robust property called ***adversarial collision resistance***, among other properties.

Must respect other property than non-cryptographic one:

1. **Adversarial collision resistance**: ensures that it is computationally infeasible for an attacker to find two distinct inputs that hash to the same output (i.e., a collision) even if the attacker can choose both inputs. Let H be a hash function with an n-bit output. Adversarial collision resistance can be enforced by ensuring that for any efficient adversary A, the probability that A can find distinct inputs x and y such that $H(x) = H(y)$ is negligible, i.e.,

$$Pr[A(x, y) = 1] \leq \xi(n)$$

where $\xi(n)$ is a negligible function of n, meaning that it approaches zero faster than any inverse polynomial in n as n grows.

1. **One-way function (Pre-image resistance)**: let X be the domain and Y the codomain of hash function h. For any $y \in Y$ it's hard to find $x \in X$ such that $h(x) = y$. Must be a one-way function.

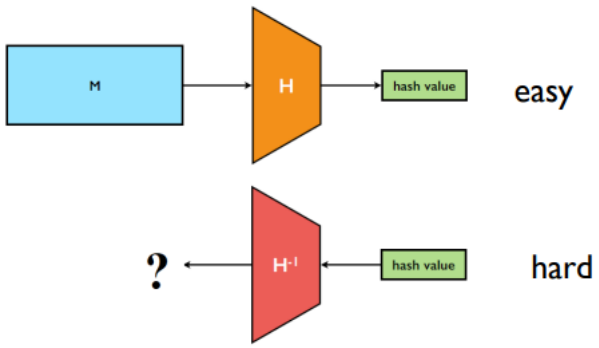
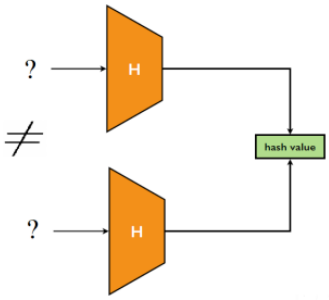
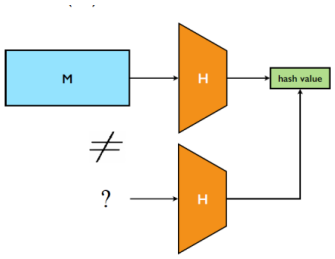


Figure 2.3: **One-way function (Pre-image resistance)**



(a) **Collision resistace**



(b) **Second pre-image resistance**

2. **Collision resistace**: it's hard to find a pair of value $x_1 \neq x_2$ such that $H(x_1) \neq H(x_2)$.

3. **Second pre-image resistance**: given M and thus $h = H(M)$ it's hard to find another value M' that $H(M') = h$. This property is also kwnown as *wak collision resistance*

4. **Hiding**: (*see later*)

5. **Puzzle-friendliness**: (*see later*)

The approaches followed for *attacking* an hash function:

- Cryptanalysis involves exploiting logical weaknesses in the algorithm
- Performing *brute force* attack: in cryptography it corresponds to an exhaustive search that might be used when it's not possible exploit other mechanism. The strenght of a hash function against brute force attacks depends on the length of the hash code produced by the algorithm: for a hash code of length n, the level of effort is *proportional* to the value in table 2.5.

So given a m-bit hash function, the attacker needs $2^{\frac{m}{2}}$ brute force computation to find a collision:

- MD5 is $128/2 = 64$ bits security
- SHA-1 is $160/2 = 80$ bits security
- SHA-256 is $256/2 = 128$ bits security
- SHA-512 is $512/2 = 256$ bits security

2.1.3 Hiding

To avoid that the attacker is able to guess the input domain, a solution is to pick a **random integer** R of 256 bits from a distribution with high min entropy (*no particular value is more likely then others*) and append R to the original input. The input space becomes extremely hard to enumerate (2^{257} possibilities). The property can be defined formally as:

"*A hash function H is said to be hiding if when a secret value R is chosen froma probability distribution that has **high min-entropy** then given $H(R||x)$ it is infeasible to find x.*"

Referring the definition, x is the **nounce** in Blockchain corresponding mechanism.

Preimage resistant	2^n
Second preimage resistant	2^n
Collision resistant	$2^{n/2}$

Figure 2.5: Number of bit to for each property

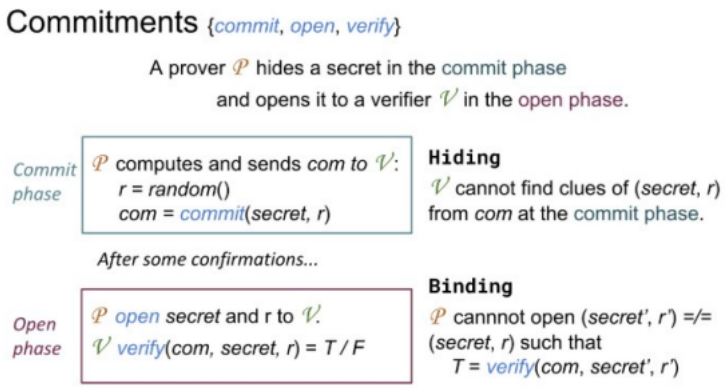


Figure 2.6: Commitment scheme for hiding

Commitment scheme (Hiding)

As briefly sketched in 2.6, it allows to commit to a value and reveal it later: seal a value in an envelope and put that envelope out on the table where everyone can see it. An implementation of commitment scheme by using hash function, in an exchange scenario between Alice (A) and Bob (B), is the following:

1. $A \rightarrow B : h_A = H(RA||paper)$
2. $B \rightarrow A$: scissors
3. $A \rightarrow B : RA, paper$

At the end of the protocol Bob need to verify that the h_A sent by Alice is equal to $H(RA||paper)$ so if the values agree Bob will known that Alice has not cheated.
This scheme allows Bob to not determine priorly that Alice has committed to the value Paper because he does't known the random value RA used (**Hiding property**) and he is unable to invert the hash function (**Pre-image resistance property**). As soon as Bob sends the value scissors to Alice, she knowns she has lost but is unable to cheat: she would need to come up with a different value of RA, say $R0A$, which satisfies:

$$H(RA||paper) = H(R0A||stone)$$

but this would mean that Alice could find collisions in the hash function: this will not happen if the hash function guarantee the **Second pre-image property**. The overall scenario is pictured in 2.7

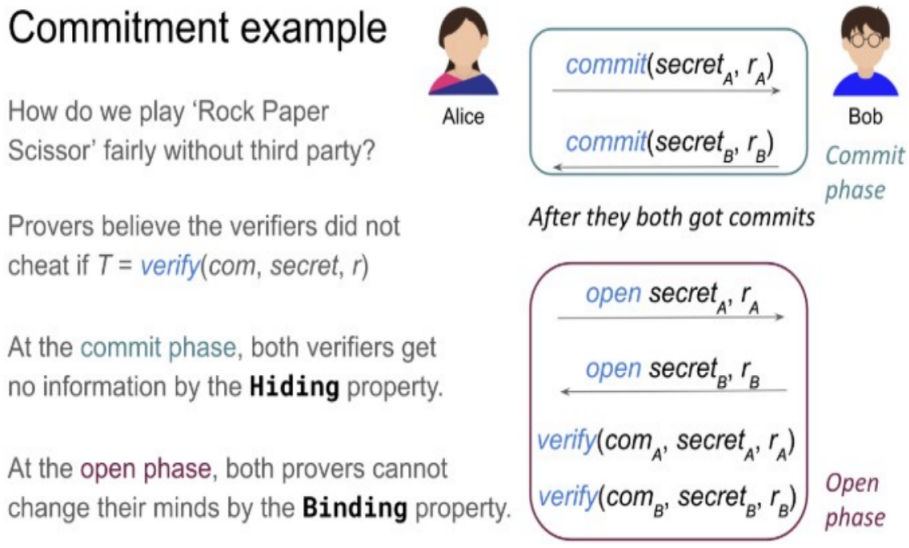


Figure 2.7: Commitment scheme between Alice and Bob

Search Puzzles

An hash/search puzzle consists of:

- a cryptographic hash function H
- a random value r
- a target set S
- a solution of the puzzle is a value x, such that $m = r||x$, $H(m) \in S$. Based on ***partial pre-image attack***, you have to find a part of the input such that the output belongs to a set (*not a single value like in the pre-image attack*) as showed in 2.8.

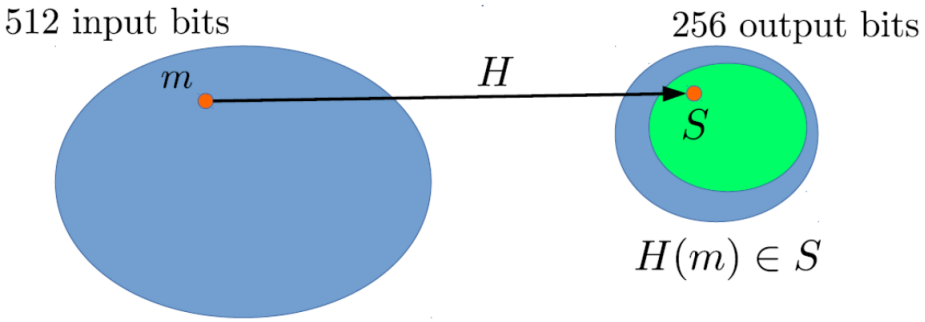


Figure 2.8: Solution on input/output set for puzzle

The difficult of may be tuned by defining the size of the set S: if it's large, the puzzle is less difficult. For example, in Bitcoin, m is defined by the number of leading zeroes of SHA-256.

The **Puzzle-friendliness property** can be defined as: *"A hash function 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's infeasible to find x such thath $H(k||x) = y$ in time significantly less than 2^n.

This property implies that **no solving strategy to solve a search puzzle is much better than trying exaustively all the values of x**.

Cryptographic hash applications

The peculiar class of hash function already defined allows to implement an **anti-tampering** property of the information by usin the hash value as the *checksum* to check if the data is changed or modified. To recognize if a content C is the same of a contwent $C1$ that we saw before it's necessary to remember the hash of $C1$, compute the hash of C and compare the two hashes so if they're equal, the content has not been tampered. This property is also embedded in the Bitcoin blockchain to store transaction ledger in a P2P Network.

2.2 Data structure for DHT & Blockchains

In the following section we'll address the following data structure (*also used for DHT and IPFS*): *Hash Pointer, Bloom filters, Merkle trees, Trie, Patricia Merkle Trie.*

2.2.1 Hash Pointers

An *hash pointer* is a pointer to where the information is stored using a cryptographic hash on the information itself. Starting from a given pointer we can ask to get the information back or verify that it hasn't changed.

The key idea is to build data structures with hash pointers embedded: in the **blockchain** view, a list is linked with hash pointers by computing the hash to a block, hash again the entire block with its own hash pointer:

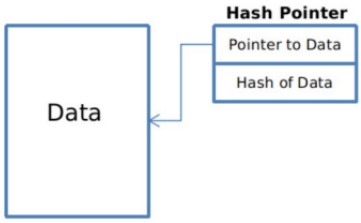


Figure 2.9: Single hash pointer

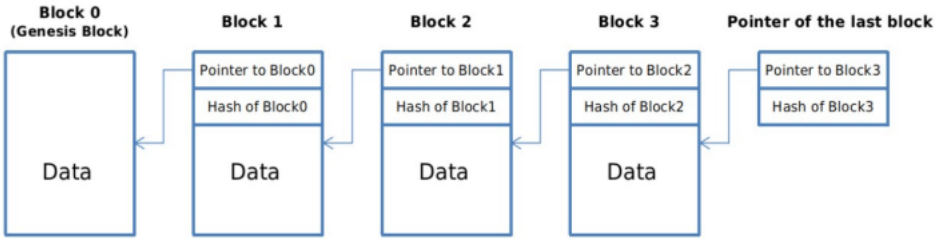


Figure 2.10: Hash pointer with multiple blocks

If someone *tampers* the k^{th} block of the chain, the hash of block $k+1$ is not going to match up because the **hash is collision resistant** so an adversary cannot tamper the data so that its hash is the same of the data before tampering.

Some **use cases** are useful applying this schema, like:

- **tamper-evident log**, a basic data structure in Bitcoin and Ethereum
- in PoW-based blockchain, the block contains also the proof that PoW has been succesfully executed
- if data is changed, PoW has to be re-executed for all the blocks
- computationally infeasible

2.2.2 Bloom filter

Consider the set $S = \{s_1, s_2, ..., s_n\}$ of n elements with n very large. Usually we want to define an efficient data structure that support queries like "k is an element of S" also called as **membership queries**: a given function f returns value $true$ or $false$ according to the presence of k in the given set S.

The Bloom filter differently allows to provide an *approximate solution* to the set membership problem 2.11:

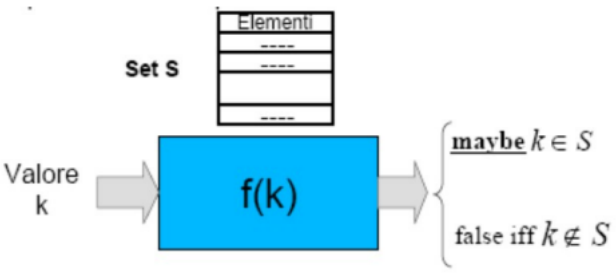


Figure 2.11: Membership problem sketcged

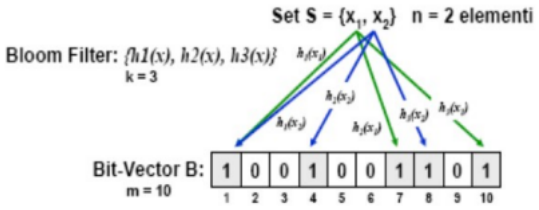
The problem is translated into choosing a representation of elements in S such that the result of the qery is computed *efficiently* and the space for the representaiton of the element is reduced. The result may be approximated to save space and arises the possibility of returning **false positives**.

Building a Bloom Filter Given a set $S = \{s_1, s_2, ..., s_n\}$ of n elements, a vector B of $m >> n$ (*generally* $m > n * k$) bits, $b_i \in \{0, 1\}$. Choose k ***independent hash functions*** $h_1, ..., h_k$ such that:

$\forall i.h_i : S \to [1..m]$, h_i returns a value *uniformely distributed* in the range $[1..m]$. Construct the filter $B[1..m]$ such thath:

$$\forall x \in S : B[h_j(x)] = 1, \forall j = 1, 2, ..., k$$

Note that a bit in B may be target for more than 1 element:



Look up To verify if y belongs to the set S mapped on the bloom filter, apply k hash functions to y so we can determine that $y \in S$ if $B[h_i(y)] = 1, \forall i = 1...k$: if at least a bit is equal to 0, the element *does not belong to the set*. The problem of the **false positives** can occur if all bits are positive because another element or some combination of other elements could have set the same bits.

Compute the probability that, after all the n elements are mapped to the vector, a specificbit of the filter (*of size* m) has still value 0:

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

The approximation is derived from the definition of e:

$$\lim_{x\to\infty}(1 - \frac{1}{x})^{-x} = e$$

A percentage of $e^{\frac{-kn}{m}}$ bits are 0, after its construction. If we consider an element *not belonging* to the set, by applying the k functions, a false positive is obtained if all the k hash functions applied to that element return a value equals to 1 and this happen with probability of:

$$(1 - e^{\frac{-kn}{m}})^k$$

The **probability of false positives** depends on two parameters: - $\frac{m}{n}$: number of bits exploited for each element of the set - k: number of hash functions If the first parameter is set:

- decreasing k increases the number of 0 and hence the probability to have a false positive should decrease 2.12
- increasing k increases the precision of the method. Hence the probability of false positive should decrease 2.13. Fixed the ratio $\frac{m}{n}$, the probability of false positives first decrease, then increases, when considering increasing values of k, as shown in 2.12.

  Let us now suppose that k is fixed, the probability of false positives *exponentially decreases* when m increases (m *number of bits in the filter*). For ***low values*** *of* $\frac{m}{n}$ (a few bits for each element), the probablity is higher for large values of k, as shown in 2.13

A bloom filter becomes effective when $m = c * n$, with c constant value.

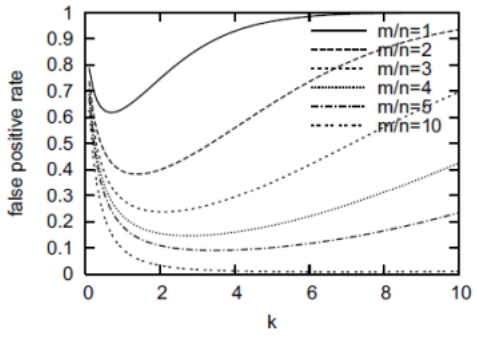


Figure 2.12: undefined undefined

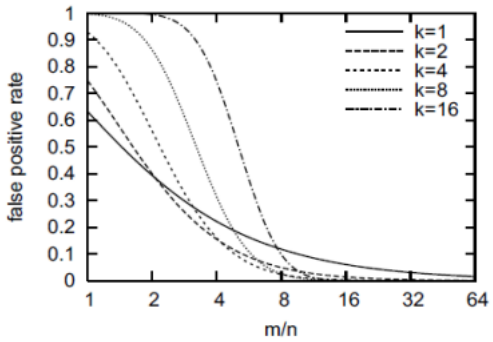


Figure 2.13: undefined undefined

Other operations

- ***Union***:Given two Bloom Filters, B1 e B2 representing, respectively, the set S1 and S2 through the same number of bits and the same number of hash functions, the Bloom filter representing $S1 \cup S2$ is obtained by computing the bitwise OR bit of B1 and B2
- ***Delete***: note that it is not possible to set to 0 all the elements indexed by the output of the hash functions, because of the conflicts. This operation is supported in a different version called *Counting Bloom Filter* in which each entry is a counter (*instead of a single bit*): at insertion time the counter in incremented while it's decremented at deletion.
- ***Intersection***: given two Bloom Filters B1 and B2 representing respectively, the sets S1 and S2 through the same number of bits and the same number of hash functions, the intersection of the Bloom filters obtained by computing the bitwise and of B1 and B2 and approximates $S1 \cap S2$.

Bloom Filter usages The set S mapped on a bloom filter may be a set of ***Bitcoin addresses***: light-weight mobile nodes do not store the blockchain and build only the bloom filter with the addresses they are interested in. They send the BF to a full node allowing bandwidth saving and privacy. The full node receives a block B of the blockhain and uses BF to check if some address of B belongs to BF. In ***Ethereum*** is used to summarize events generated by a smart contract by finding all the tokens sold by a specific user in a block of 500 transanction. Instead of parsing all transaction, query a Bloom Filter (*or ***log bloom***) in the header of a block for the presence of that users and perform a search in the block only if a match is found.

2.2.3 Merkle Hash Tree

A merkle hash tree it's a data structure that allows to *summarize* a big quantity of data with the goal of verifying the correctness of the content. It's a **complete binary tree** of hashes built starting from a initial set of data:

- i^{th} leaf stores the has of h_i of f_i (*i^{th} *file*)
- an internal node contains the concatenations of the hashes of the sons of the node
- the last hash stored in the root is called ***Merkle Root Hash***. The structure obtained is pictured in 2.14

Suppose have $n = 8$ files $f_1 ... f_8$ and a collision resistant hash function H: go on hashing every two adjacent hashes by applying $H(x, y) = H(x|y)$ where | indicate concatenation. A ***Collision-Resistant Hash Function*** for **Merkle Hash Tree (MHT)** takes n input $(x_1 ... x_n)$ and outputs a *Merkle root hash* $h = MHT(x_1, ..., x_n)$.
The MHT has an important proof-property: if a *verifier (Alice)* only known the Merkle root has h, the *prover (Bob)* can give Alice one of the values x_i and convince ALice that is was the i^{th} input used to compute h: to convince her, Bob gives Alice an associated ***Merkle proof*** without shoging all the other inputs. So if a merkle proof says that x_i was the t^{th} input used to computed h, no attacker can come up with another Merkle proof that says a differnet $x_i \neq x_j$ was the i^{th} input used in MHT.

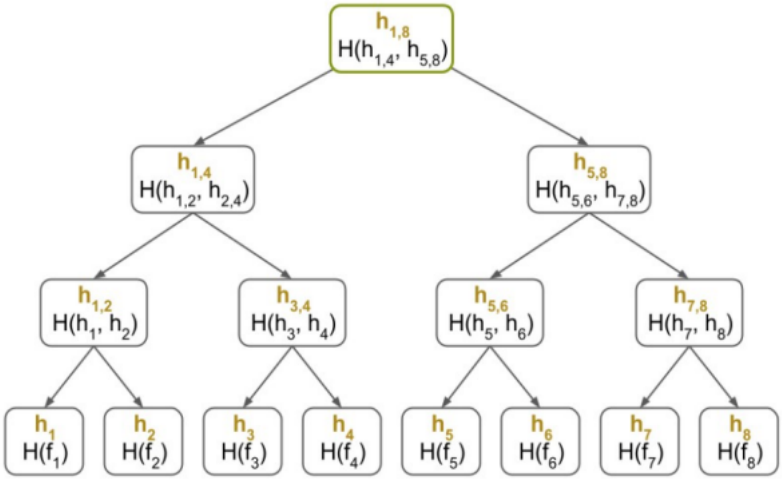


Figure 2.14: Merkle hash tree chains

Merkle Proof The key idea is to have a given file f_i and to download a small part of the Merkle tree called *Merkle Proof* of **Membership proof**: it contains the subset of hashes of the Merkle tree, that, together with f_i, allow to recompute the root hash. A proof is given for a specific *leaf node* and is comprises the leaf's siblings, and the siblings of each node on the path from the leaf to the root, as shown in 2.15. The proof enables you to verify that f_i was not modified by checking if the root hash obtained by the proof matches the Merkle root.

To verify the proof, *"fill the blanks"* by computing the missing hashes in the dotted lines, then check if the computed Merkle root you kept locally is equel to the Merkle root obtain from the computation.

*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.

- ***Proof (by intuition)***: if the Merkle proof is verified, you were able to recompute the root hash by using f_i as the i-th input and the Merkle proof as the remaining inputs. If the proof verification had yielded the same hash but with a different file as the i-th input, this would yield a collision in the underlying hash function H used to build the tree. *Is not possible if H is collision resistant.*

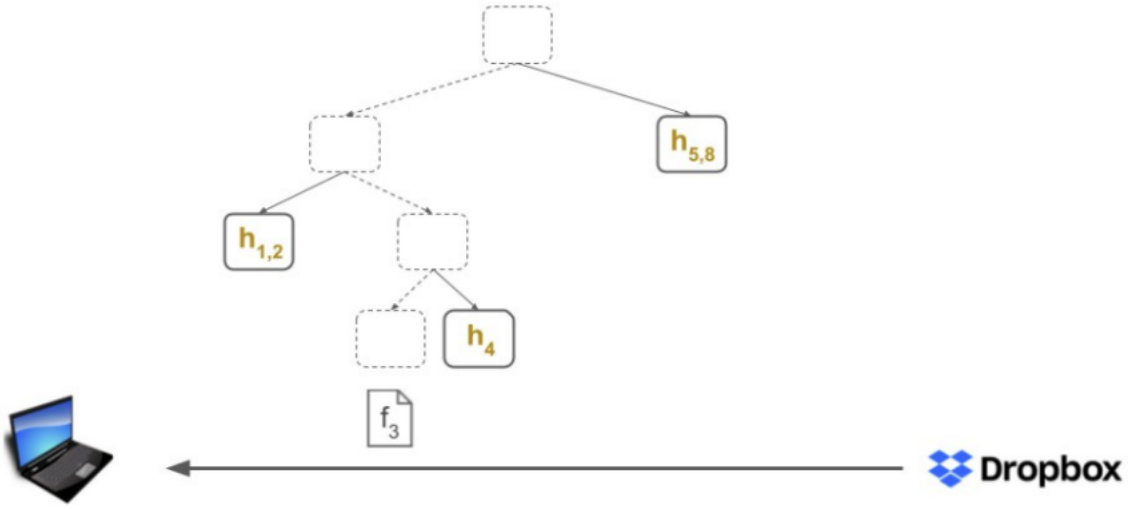


Figure 2.15: Merkle proof sketched

Rationale behind large merkle proofs The main reason to not simply use the has of the files to check their integrity is that the hashes h_i are much smaller than the files themselves: can be done by storing the hash $h_i = H(f_i)$ of every file, rather than the Merkle root so when you download a file, only compute the hash $y_i = H(f_i)$ and check thath $h_i = y_i$. The merkle proof solution is suitable in case of a huge number of files as in *certificate transparency project* which stores millions of digital certificates for HTTPS websites. The Merkle Tree may have hundres of millions of leaves and it's designed to scale to billions by mantaining a $O(log(n))$ size.

Merkle Tress in BitTorrent In BitTorrent can help ensuring that *data blocks received from other peers are received undamaged and unaltered*: a trusted third party, like the site indexing the `.torrent` stores the *root hash* and the *total size of the file and the piece size*. So a **peer** receives a piece and a *Merkle proof* for it, calculate the has of that piece, request the root hash from the trusted site and by combining this information the client can recalculates the root has of the tree and caompares it to the root hash it received from the trusted source.

Merkle Tress in Bitcoin In a transaction scenarios between Alice and Bob, if Alice need to verify if the transaction has been inserted in the blockchain and don't want to download the entire blockchain, can use different solution:

- *Insicure-polling*: Alice simply asks the nodes storing the full blockchain if Bob's transaction has been inserted in the blockchain. Bob can lie by sending a fake transaction.
- *Merkle-based solution*: build a Merkle tree of the transactions in a block, a full node sends a Merkle Proof to Alice so she can finds the root of the Merkle tree in the block headers. In this way she must download only the block headers and not the entire blockchain.

Merkle Tress Complexity Summary n is the number of data items, m hash size:

- size of the Merkle tree: $O(n)$
- size of the Merkle root: $O(m)$
- size of the authentication path: $O(mlog(n))$
- $log(n)$ hashes are sufficient for checking each data block

2.2.4 Trie

The Trie data structure is a tree-like data structure that is commonly used to store and retrieve strings efficiently. In a Trie, each node represents a prefix or a complete string, and each edge represents a single character. The key operations of a Trie are inserting a string, searching for a string, and deleting a string. The complexity of these operations depends on the length of the strings being processed and the size of the character set. The main operations are

- **Insertion** in a Trie involves traversing the tree until the end of the string is reached, and then adding a new node for each character that does not already have a corresponding child node. If the string is already in the Trie, no new nodes are added. The time complexity of insertion is $O(m)$, where m is the length of the string being inserted.
- ***Searching*** in a Trie involves traversing the tree from the root to the node corresponding to the last character of the string. If the path does not exist in the Trie, the search fails. The time complexity of searching is $O(m)$, where m is the length of the string being searched for.
- **Deleting** a string from a Trie involves first searching for the string, and then removing the nodes that correspond to the string's characters. If the string is not in the Trie, no nodes are removed. The time complexity of deletion is also $O(m)$, where m is the length of the string being deleted.

The space complexity of a Trie is determined by the number of unique strings that are stored in it. The worst-case space complexity is $O(n * m)$, where n is the number of strings and m is the length of the longest string. However, in practice, the space complexity of a Trie is usually much less than this, since many strings share common prefixes and can be stored using shared nodes.

Patricia Trie

Patricia stands for "*Practical Algorithm To Retrieve Information Coded In Alphanumeric*". It compress long one-child branches into single node. In the 2.16, on the left a standard Trie, on the right the associated Patricia Trie.

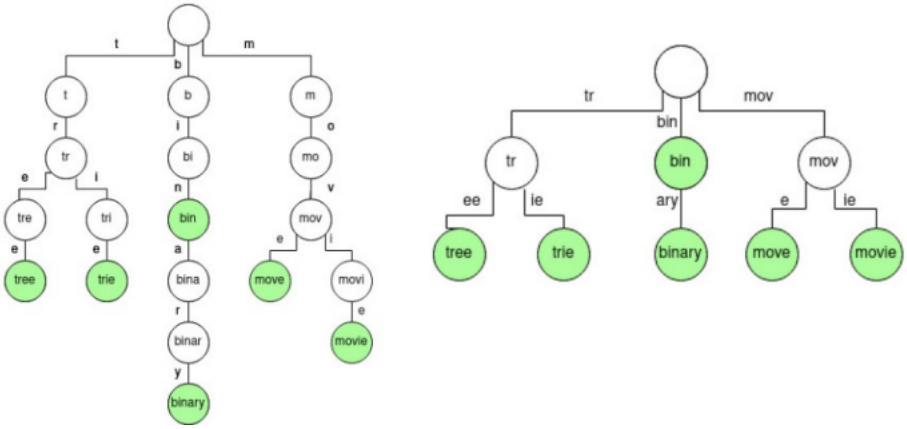


Figure 2.16: Trie vs Patricia trie

2.2.5 Patricia Merkle Trie

The **Patricia Merkle Trie** is a combination of both:

- Patricia Tries: to enable faster search of data, stores keys, while grouping their common path in a node
- Merkle Tree: to maintain data integrity and tamper proof validation It is organized as a tree with every node hashed in the sense of *Merkle proof*, grouping common paths in the sense of *Patricia tries*.

Regarding the Patricia Trie, it is not limited to string representation but can be used to store (`key,value`) pairs. Keys are string represented by the trie and the value will be stored in, which is at the end of a key path.

- **Branch node**: a node that has more than one child node. It stores links to child nodes, may also store a value.
- **Extension node**: a non-leaf node representing a sequence of nodes that has only one child. It sotes a key value that represents the combined nodes and a link to the next node.
- **Leaf Node/Value Node**: similar to an extension node. Instead of link, it stores a value.

In the example pictured in 2.17, Key-value mapping are represented in the Patricia trie `(BIN, 10) (BINARY, 8) (MOVE, 30) (MOVIE, 55) (TREE, 20) (TRIE, 48)`.

It's possible to ***"Merkelizing"*** a patricia trie by making the tree cryptographically secure, pairing each node with its hash. The root node becomes a cryptographic fingerprint of the entire data structure. The hash may be used for *lookup* in a database and to refenrence child nodes, as shown in 2.18.

The **Ethereum blockchain** store:

- the state of the contracts on the blockchain
- a set of transactions in blocks

The state is a combination of `key/value` pair, may be that:

- `Key` is the address of an account, `value` is the account balance
- `Key` is the address of a transaction, `value` is the transferred amount Ethereum uses Merkle Patricia tries to store all the information on the blockchain.

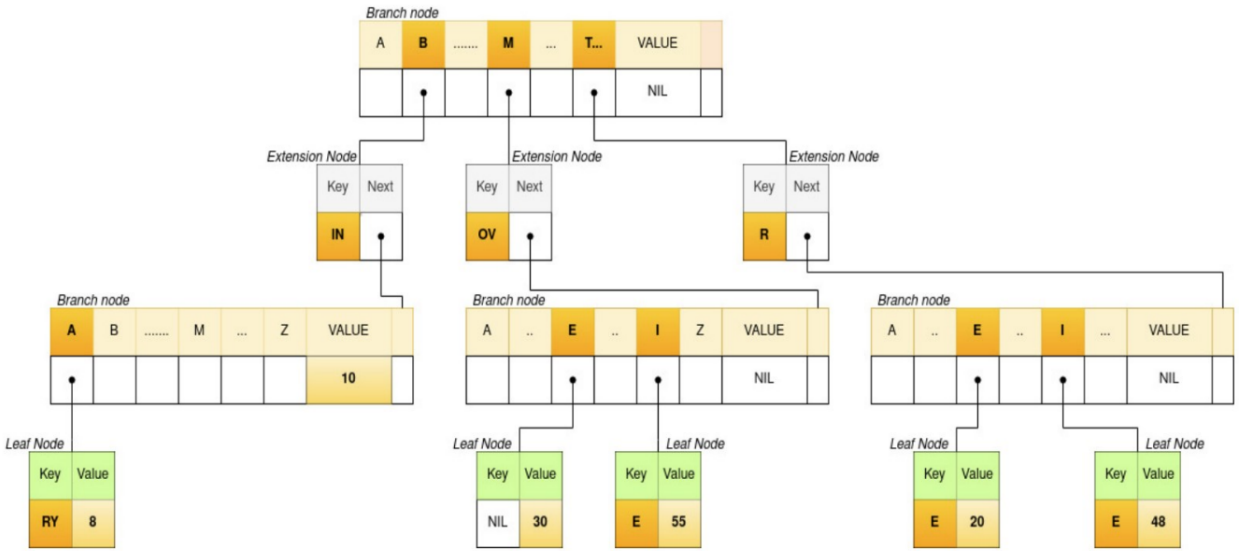


Figure 2.17: Patricia Merkle Trie example

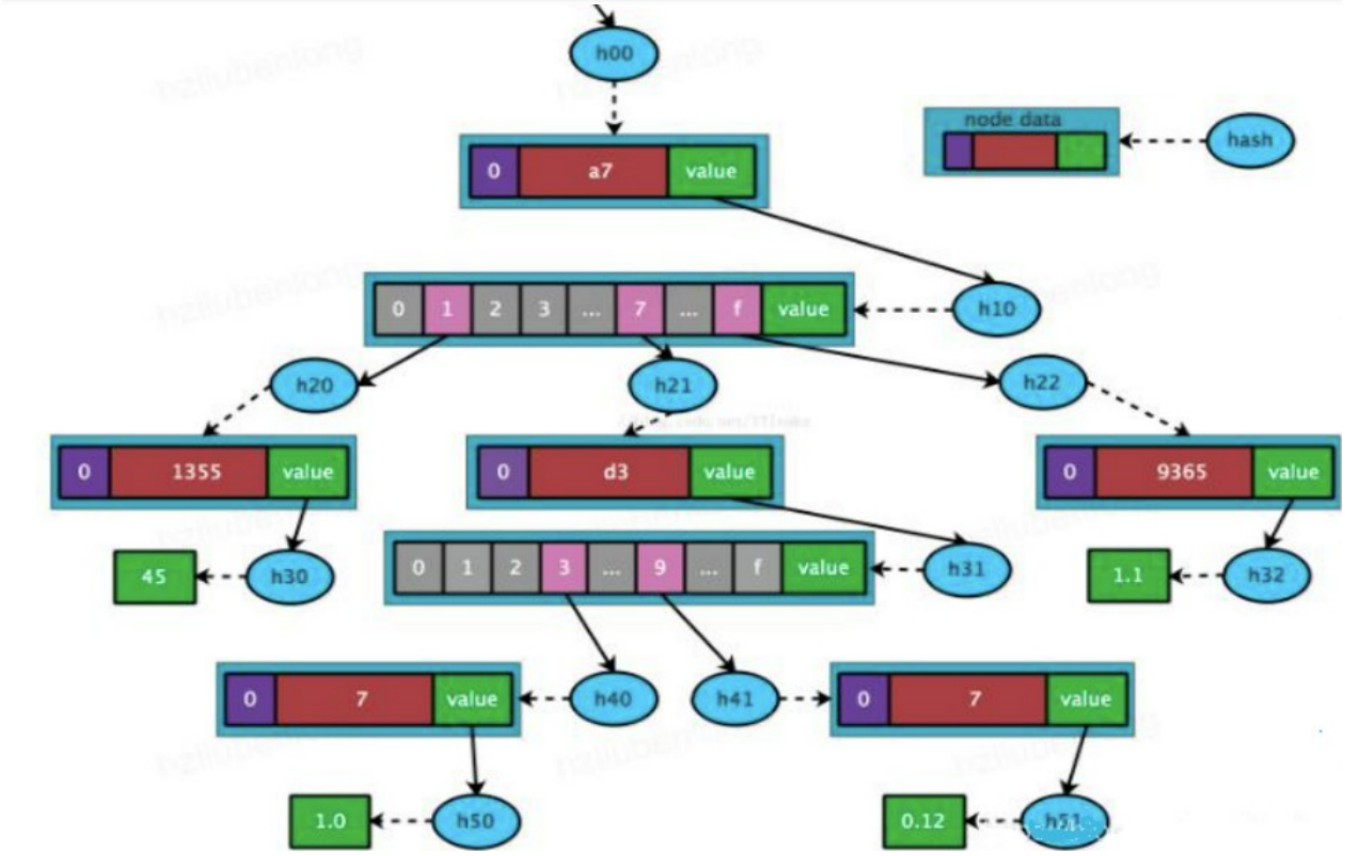


Figure 2.18: Merklezized patricia trie: root act as a fingerprint for entire data structure

The nodes of the Patricia Trie are *multi-element, structured records*, while hash function is applied to binary string so it requires some kind of ***"serialization"*** to encode structured records as byte array (*like JSON, but more efficient*). Different techniques are used to encode a variety of data structures; two main techniques used by Ethereum are:

- ***Hex Prefix Encoding (HP)***: HP encoding is used for encoding/decoding Keys (Paths).
- ***Recursive Length Prefix (RLP)***: RLP is used for encoding/decoding Values, corresponding to keys (which may be structured values

Chapter 3

Bitcoin

3.1 Blockchain introduction

We'll introduce first basic concepts like **ledgers, consensus in distributed environment, tamper freeness, proof of ownership, permissioned and permissionless blockchain**.
A **blockchain**, as a general idea, is a *replicated ledger* on the nodes of the P2P network: it's consistent and immutable, despite the ledger can differs each other for a short period of time but they'll converge. The immutability derive from the nodes approvation, guaranteeing tamper freeness. Inside the blockchain there can be at most 3 types of data:

1. Hash of the current block
2. Hash of the previous block
3. Associated data within the current block

The data, in case of transactions, are related to the source, destination and the amount. The two hash represent the chain, allowing chaining each block with the previous one. The **tamper freeness** guarantee that if changing one hash causing changing the hash of the following block but this does not imply to recompue some hashes, but it's necessary to find a value of the new hash that solves the Proof of Work. This property is also guaranteed by the distributed consensus algorithm.

Ledger

A ledger store operations and mantains the order of operations. It have properties like:

1. *Append-only* list of event
2. *Tamper-proof*: inside an approved block is not possible to modify or remove a data. This guarantee also auditability.
3. *Consensus*: everyone agrees on the content It can be used in each application that require a log of events.

if the ledger is organized as a list of blocks it forms a blockchain but other topologies are possible like *graphs*. Assume as hypothesis that a block contain only a single operation (*not true for Eth or Btc*): the linearity of the structure and the consensus algorithm allows to avoid redundancy problems (*like the Double Spending probem*), ensuring that the information stored are consistent each other.

The **consensus** is the mechanism which defines:

1. who decides which operation will be added to the blockchain. This also implies select an *honest* node among the candidates.
2. which operation among those to be confirmed, will be added

The consensus so represent an **agreement on the same value**: the nodes agree on one of the node's input and this proves the **validity** so, agree on someone's proposal. The **tamper freeness** is also guarantee through hash: computing the hash of each entry and storing in each entry the *predecessor's hash*. If one entry is *tampered*, there is thhe need to recompute the has of all the following one and this operation must be ***computationally hard*** so the hash of proof of work must be re-associated with the block.

This poses several challenges, like:

- Mantain consistency in opresence of different network jitter, delay, eyc
- Some nodes can **cheat** so it's necessary to create **bizantine parties** (*see Byzantine Fault*) A classical result is that if the majority of the node is honest the consensus algorithm works well: the definition of *majority* change with the protocol used. As a **honest majority** we refer to the nodes which follows correctly the protocol by implementing and executing the consensus by voting: broadcast every operation on the network and collect all the votes properly.

The **double spending** problem is correlated to the consensus algorithm: the problem cannot be avoided totally and it depends on the notion of *"majority nodes"*, as we will see.

Sybil attack It consists in inject **multiple fake indetities** into the network: it's necessary to register with the DHT multiple times. This type of attack can have as a goal to control data replica, attack routing or disrupt network connectivity. To guarantee that a single node cannot impersonates multiple logical identies some solutions are:

1. *Proof of Work*: require computational power, impling that the identity is not a single one
2. *Proof of Stake*: require stake
3. *Certified Node-IDs*: requires a central authority

Those methods poses a solution to define the majority in a context where everybody can easiily join the network and assume multiple identity. Using the **computing power controlled by each identity** allows to create multiple identities but are not usefull if in a single node/computer/controlled only by one subject.

Proof of Work As an example, the *Proof of work* is like a lottery to choose which node will decide the next block and the tickets are the production of proof of work by solving a complex computational problem. The nodes that win the lottery is paied when endorses validity and decies which is the next node of the blockchain.

If two nodes "*win*" simultaneosly, so they have both the right to choose the next block this causes a **fork**: the two winner attach their new block to the blockchain, so those blocks are linked in parallel with the previous block. This scenario is infrequent but not impossible: it's solved when later winner decide which forks win, prouning the other.

Proof of Ownership In case of an *ICO - Initial Coin Offer* there is the necessity, after registering tokens on the blockchain, that the ownership of those tokens is tracked for each transaction executed on them. Because there is not centralized certification authority, a completely distributed solution is using **Asymmetric Key Cryptography**: generate the pair of key and anyiine who knows the private key matching the public key of Alice also owns the Alice tokens. The *public key identifies* the owner of the token publicly, the effective ownership is demonstrated by the propriety of the *private key* by signing the transfer operation. The transactions are registered on the ledge along with the signed transaction that can be verified by the receiver.

Blockchain access types

Two main type:

1. **Permissionless blockchain**: the property of a **permissionless blockchain** means that anyone can partecipate to the consensus of the blockchain, participating to the governance of the blockchain itself. Of course, in those type of blockchain, other operations are allowed like reading transactions etc. There is no central authority and the partecipation to the consensus is based on reward.
2. **Permissioned blockchain**: the involves parties have identities (*human have password, sensors/automated nodes have keys*) and there is the guarantee to avoid the Sybil attack.

Based on the specific case of a supply chain, we can **restrict the access** to only the actors involved in the process of consensus: they can also be automated actors/nodes.

An interesting proposal is the **Practical Byzantine Fault Tolerance** (*see MIT - PBFT*) consensus algorithm for small number of nodes.

3.2 Transactions and scripts

Bitcoin is a Purely P2P version of digital cash without a controlling authority, no server and no banks as intermediary. It's also *permissionless*, thus without a regulator and *censorship resistant*, avoiding the frozen funds problem. Initially was basically *free*, with neglectible transaction costs (*today the costs skyrocketed*) and *borderless* without geographic/national limits.

The proposed system allows *disintermediation* and *pseudo-anonimity* because a transaction is performed without identity disclosures but still exists methods to discover the identity (*at least partially*) behind a transaction.

3.2.1 Decentralized Identity Management

The identity is represented in a complete decentralized system as a cryptographic information, usually a *random key-pair values*:

- *sk*: private (secret) key
- *pk*: public key. Is the public *name* of a user, despite usually as a public name is used *Hash(pk)* The pair allows only the owner of *sk* to control the identity: a transaction signed with a signature *sig* such that *verify(pk, data, sig) == true* so *pk* has generated the transaction because the *pk* is public and can be associated to a given user (*identified by Hash(pk)*).

The main cryptographic tool used in Bitcoin is *ECDSA - Elliptic Curve Signature Algorithm* (*secp256k1 curve*): it's used to sign transaction with the private key and to verify the signature of the transactions (*having the corrisponding publick key*). ECDSA is not a *post-quantum algorithm*. Bitcoin does not encrypt any information: keys are used only to prove the ownership of a given transaction that is public.

Keys and addresses The **privte key** K is a umber, usually picked at random: the ownership control over the private key is fundamental to control all fund associated with the corresponding BTC address. From the private key K, it's possible generate the *public key* Q through elliptic curve *multiplication* operation (*a one-way function*). From Q is possible to generate the BTC address A using another one-way function. The address generation is sketched in 3.1:

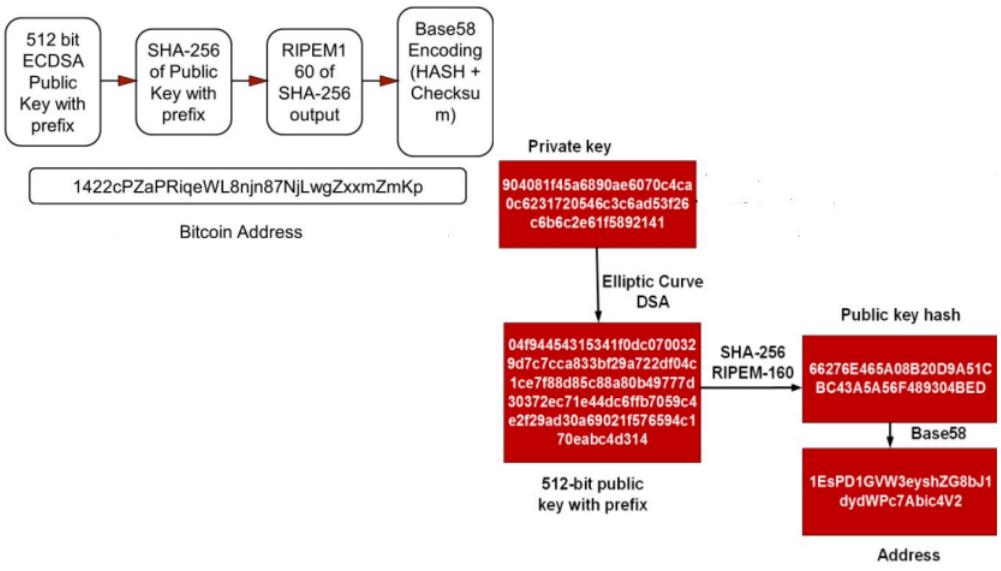


Figure 3.1: Address generation

```
alphanumeric = 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
base58       =  123456789ABCDEFGH JKLMN PQRSTUVWXYZabcdefghijk mnopqrstuvwxyz
```

Figure 3.2: Base 58 vs classical alphanumeric

Using the ***base 58*** *(3.2)* implies that the number of char left with when using all the characters in the alphanumeric alphabet (62), byt remove all the easily mistakable characters, avoiding using *0,O,l,I* that can appear identical when displayed in certain fonts. The advantages are that a large of character still remain and cna represent large numbers in a shorter format.

3.2.2 Payment

The general workflow is sketched in 3.3, where:

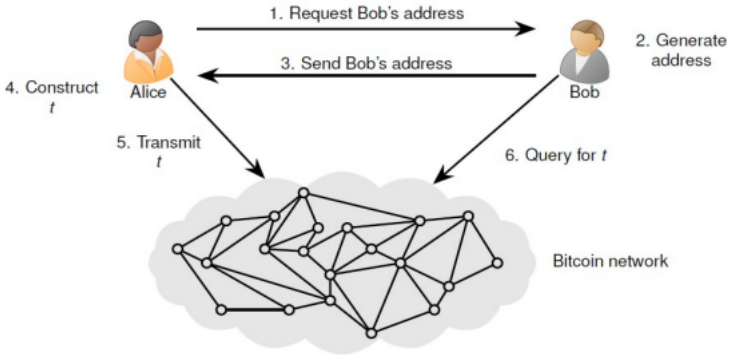


Figure 3.3: undefined undefined

1. Bob shares its address A with Alice
2. Alice generate a transaction t which pays to A and broadcast t on the P2P network
3. Miners collects broadcasted transaction into a candidate block
4. Bob wait for confirmation on t before providing the good

Inside a single wallet there are several addresses (3.4): it contains a list of *private-public* key and for each pair it's associated an address.

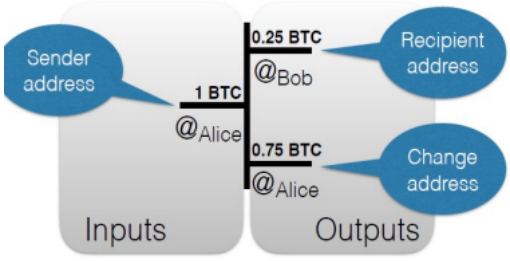
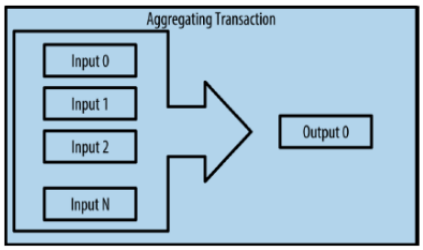


Figure 3.4: Wallet simplified

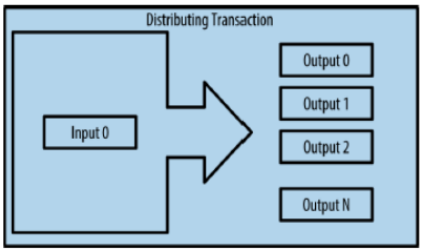
The output goes partially to Bob and partially is the *change*: usually is retuned to the same address or, to increase anonimity, is created another address associated within the Alice wallet to return the change. All the input must be consumed, so input canot be splitted to let a change in the input address. The *transaction fee* is $\sum input - \sum output$: the difference between the two represent the fee. The higher the fee the fastest the registration and validation protocol by the miners.

We can also have **multi-input transaction** (3.5a): having several addresses in the wallet and having the need to aggregate the several amounts in different addresses to only one address. Each input can also

be owned by *different users*: the joint payment must be signed by all the input's owners to produce and validate the transaction. This type of transaction is also susceptible to the **dustying attack** (*see later*). Another type of payment is the **mono-input transaction** (3.5b) with one input and multiple output: it allows to distribute the value in input to multiple recipients.



(a) Multi-input transaction



(b) Multi-output transaction

Unspent TXransacton Output Model

The transaction model explained is called ***UTXO Model*** or **Unspent TXransacton Output Model**: the *unspent output addresses* are addresses that contains bitcoins that are not spent in any transaction. According to this model, each transaction input is linked to the output of the previous transaction: each transaction generate a new UXTO, which isincluded in the user's wallet and is available to spent. Each UXTO (*so it's no more an UXTO*) is spent if it's linked to the input of a the next transaction, as sketched in 3.6.

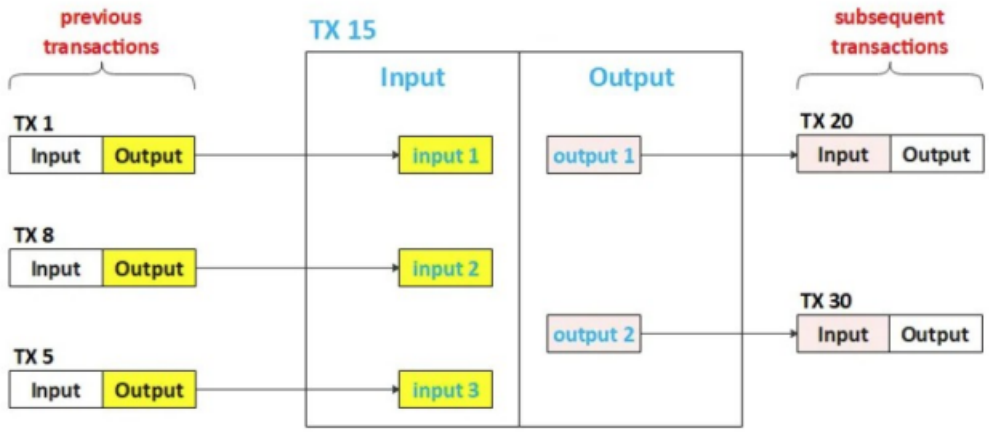


Figure 3.6: UTXO model

Usually only the last UTXO not spent is tracked to avoid to mantain in memory all the blockchain's blocks. Each UXTO *locks* the newly generated UXTO to the new owner's public key: if the owner decides to use the UXTO in a new transaction, it must *unlock* the funds with the private key, signing the transaction. Who receive the transaction, became the owner of the unused UTXO. Another way to visualize the model is pictured in 3.7:

In figure 3.7, the red circle cannot be used as transaction input, the green circle are the active addresses and new transaction can only use this as inputs. Only the unspent output are significative and mantained by the UXTO: to find the **balance of one address** is necessarily to sum the unspent UTXO of a given wallet, summing all the amounts in the different addresses in the same wallet.

The concept of a user's bitcoin balance is a derived construct created by the wallet application: the wallet calculates the user's balance by **scanning the blockchain and aggregating all UTXO belonging to that user** so an address balance is the sum of bitcoins in unspents outputs.

Scripts

It's contained inside a transaction: it's a simple program written in a simple programming lanaguage and it's **used to verify that the signature in the transaction** correspond to the public key. It allows

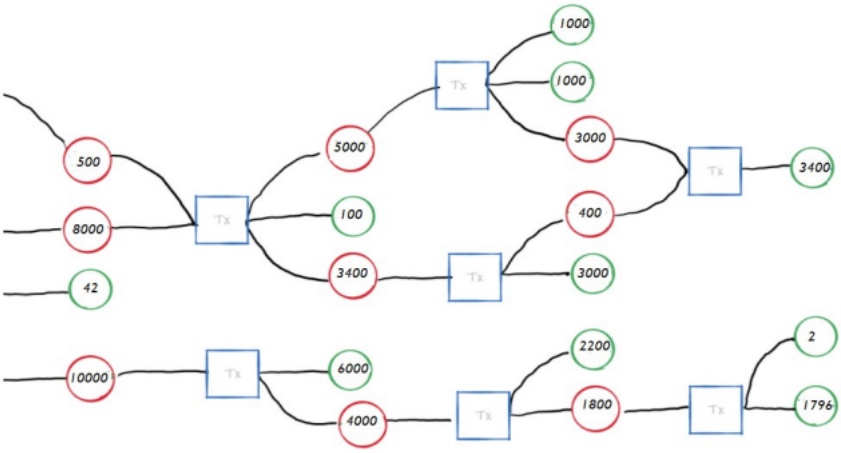


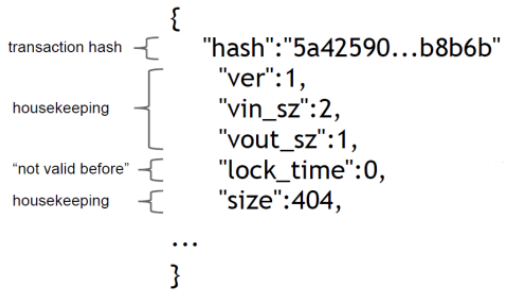
Figure 3.7: Different payments under UTXO model

to verify the ownership of the transferred coins: the code is executed by all the node of the blockchain, particularly the miners.

Here is presented a real example. The **transaction header** is reported in 3.8a. The relevant fields are:

- *hash*: report the hash of all the information contained in the transaction itself.
- *housekeeping*: to specify the protocol version, the fork, etc, allowing to use values in different way based on version. It also contains the size of input and output and the *lock_time* define the earliest time that a transaction can be added to the blockchain: if setted to 0 is immediately executed.

The **transaction body** is pictured in 3.8b



(a) Transaction header

(b) Transaction body

The *in* array is a JSON array in which each element contains a key-value pair where the key is the hash pointer to a previous transaction and index of the previous transaction's output to be spent. The *in* contains also the **unlocking script** as a *signature*. The second JSON array is *out* (3.9): contains a pair where the value is the value to be transferred in output and te **locking scripts** that contains the address (*the public key or its hash*) where that value has to be transferred.

The JSON array

Scripting language model The Bitcoin scripts is based on a ***stack-based, non-Turing complete***: this avoid to create infinite loops to avoid that the nodes who validates the transaction can enter in a infinite loop, disrupting the network.

It's **stateless** because there is no state prior to execution because all the information needed to execute are contained within the script. It's also **determinsitic** so will predictably execute the same way on different system. It's a compact scripting language, consisting of one bytecode for the *OPCODE*, allowing a total of 256 instructions: basic arithmetic, basic logic (*IF...THEN...ELSE*), special purpose instructions to support cryptography (*hashes, signature verification, multisignature verification*). So a script can be defined as *"a piece of code that verifies* ***a set of arbitrary conditions*** *that must be met in order to spend coins"*. There are several script types, from the most *simple signatures checks* that reedem

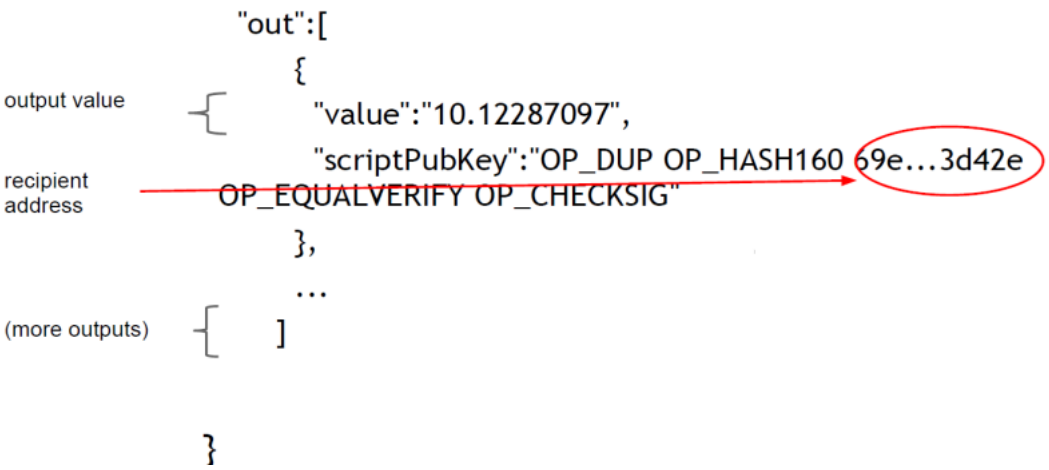


Figure 3.9: Out array content

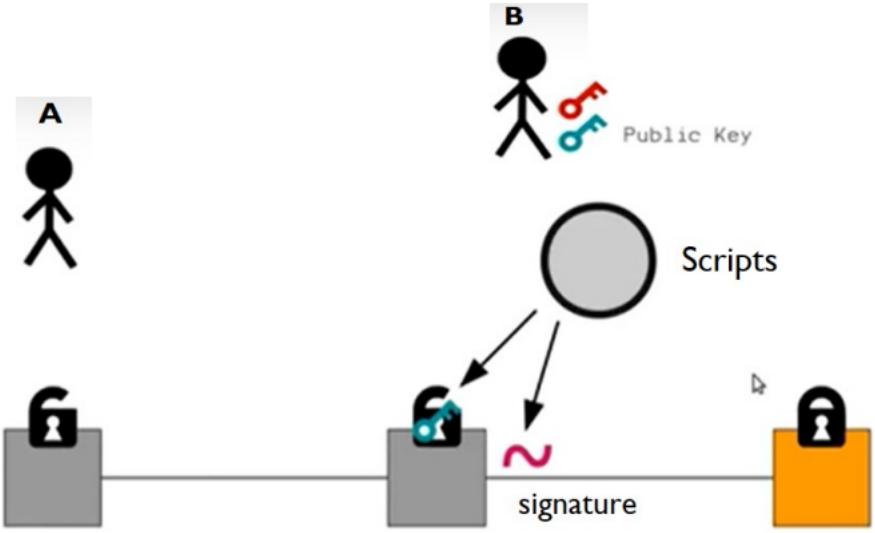


Figure 3.10: B is able to unlock using the private key (red key)

a previous transaction signing it to the most complex like *MultiSig, proof-of-burn, Pay-to-Script-Hash,* ***escrow transactions, green addresses, micro payments***.

3.2.3 Bitcoin scripts

Following the pictured example in 3.10:

- `B` notifies its public key (*green key*) to A
- `A` sends its bitocin to the public key of B (*use public key is a semplification of the B's addresses*)
- `A` unlock some of its bitcoin that were locked by a previous transaction. He creates a lock on the bitcoin sent to B (*to the public key of B*)
- Only `B` will be able to unlock the received bitcoins using its *private key (red key)*

  The script is executed in the *locking* and *unlocking phase*: referring the previous example, it's executed when the signature of `A` is verified in the transaction to trasnfer it to `B` (3.11).

The script complexity can change both in terms of key involved in the signing process and operations itself. The simplest example of script it **Pay to PubKey (P2PK)**: in this case, the locking script is inside the output of the transaction while the unlocking script is given as an input to the transaction. The figure 3.12 describe the script execution for this scenario.

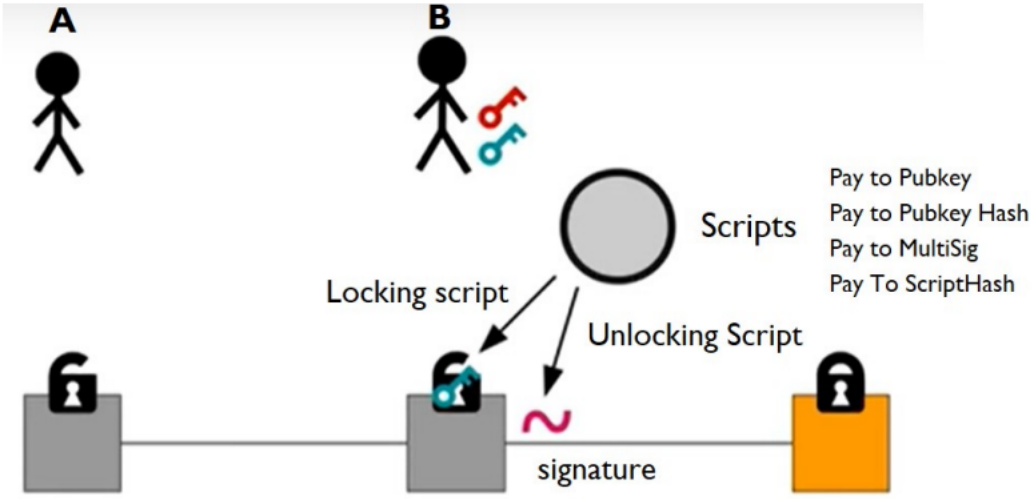


Figure 3.11: Locking and unlocking script execution

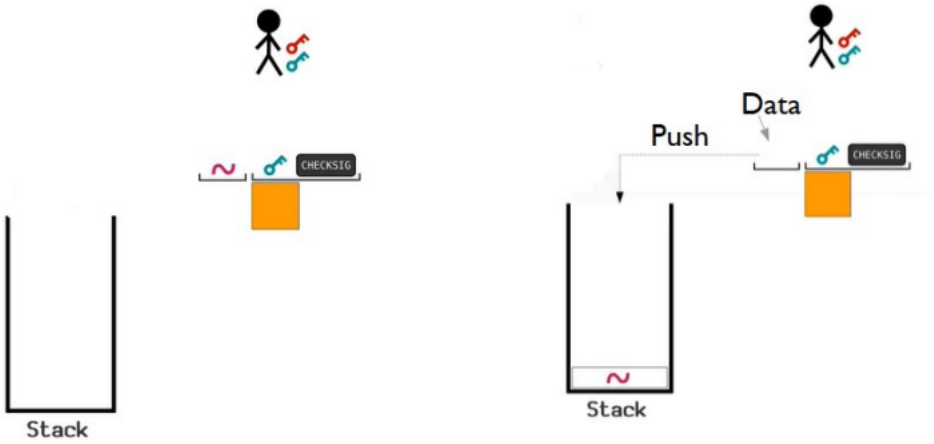


Figure 3.12: **Pay to PubKey (P2PK)**

The locking script is formed by `<Public Key> CHECKSIG`, while the unlocking script is only formed by the `<Signature>`. The execution of the CHECKSIG operation is executed by using the stack, taking the signature on the stack and veryfing that the actual owner have the rights to spend the given amount of BTC. Another type of script, the most used, it's known as **Pay To Public Key Hash (P2PKH)**: in this case, the locking script contain hash of the key, instead of the public key directly embedded into the script. The unlocking script must provide a public key which returns that hash of the address contained in the locking script and the signature corresponding to the public key. The script execution first hash the public key and compare to the hash in the locking script, chekcing if the signature is valid (*as in P2PK*).

Transaction Lifecycle It starts with the transaction's creation, then the transaction is *signed* with one or more signatures idnicating the authorization to spend the fund referenced by the transaction. It's then broadcasted on the Bitcoin P2P Network and each network node (*partecipant*) validates and propagates the transaction until it reaches every node in the network. The transaction is verified by the miner nodes and included in a block of transaction, recorded on the blockchain. From that moment, the transaction is a permanent part of the blockchain: the fund llocated to a new owner by the transaction can then be spent in a new transaction, extending the chain of ownership.

There is the pseudo-code to receive a transaction `t`:

```
for each input (h, i) in t do
if output (h, i) is not in local UTXO or signature invalid
```

```
then Drop t and stop
end if
end for

if sum of values of inputs < sum of values of outputs then
Drop t and stop
end if

for each input (h, i) in t do
Remove (h, i) from local UTXO
end for

Append t to local memory pool (waiting for confirmation)
Forward t to neighbors in the Bitcoin network
```

The algorithms does not contempalte the case for a 0 input transaction. All the nodes execute the previous algorithm when receiving a transaction, describing what is called as a **Local Acceptance Policy**: the transaction which are locally accepted by executing this algorithm ***may not be globally accepted*** so the transaction considered *unconfirmed* are added to a pool called the **Local Memory Pool** so when they will be added to the Bitcoin blockchain when they are globally confirmed. There are different local memory pools for different *unspent transaction output* in different nodes because of the **double spending problem**: this implies that the nodes will only guarantee ***eventual consistency*** by executing **Consensus Algorithms**.

3.3 Distributed Consensus

It's a procedure to reach a *common agreement* in a distributed/decentralized environment with multi-agent system. In a traditional distributed system there is a small number of nodes and fault tolerance is considered. The faulty nodes can crash or become unavailable. One of the most notable concepts is known as **Byzantine Fault**, referring to a node that start acting maliciously. The **Nakamoto Consensus**, proposed initially for Bitcoin, was based on few principles:

- **Implicit approach to consensus**: there is no voting so there is an implicit agreement where the majority wins. There is no collective message passing alg. executed by the nodes.
- **Eventual consistency**: occasionally nodes see an incosistent view of the ledger but *eventually* everyone sees the same history of the ledger. In this scenario the main assumption is that majority of the nodes must be honest.

The proposed Nakamoto consensus works in practice but it's difficult to make a theoretical proof, in fact *"the mechanism through which Bitcoin achieves decentralization is not purely technical, but it's a combination of technical methods and* ***clever incentive engineering***".

The **consensus algorithm** is necessary mainly to avoid the problem of **Double Spending**: if during the broadcasting of a transaction to all nodes another transaction occur, referring the same Bitcoin, it will be registered in the ledgers, realizing the double spending of the same amount. Even try a rollback when a node recognizes the double spending, still remain the problem of two conflicting transaction on how determine which is valid and which one will be discarded. The described scenario shows the necessity to have a consensus algorithm.

In Bitcoin, every node mantains in RAM a *temporary memory* called **MemPool** that contains a collection of all Bitcoin ***transaction awaiting confirmation***. Those transaction wait until a consensus is reached and are included in the next ledger blocks. This pool is not an *UXTO* so they're not yet registered. The *conficting transaction* may occur in MemPool and not in the ledger: whena node receive a double spend, mark it as a conflicting transaction and discard it. Of course some nodes can try to get the transaction out from the MemPool and put them onto the ledger: the final decision it's based on a **competition** between nodes. In case of **Nakamoto consensu**, the competition is implemented like a *lottery*: the process for cmpeting to add transaction from the MemPool to the ledger is called ***mining***. The winner adds the valid transaction from its MemPool to the ledger: broadcast the update version of its ledger to the neighbours by sending them the new added transaction that have been added. The nodes that receives the new added transaction must ensure that no other transaction in its MemPool are conflicting, kicking out eventually conflictingone so MemPool is like a *clearing house* for transactions.

3.3.1 Mining

The process consisting in adding a new block to the blockchain: a *network-wide competition* where every node in the network can work to try to add the next block to the blockchain. The mining process start with filling a **candidate block** witht ransaction taken from the memory pool: the node then construct a block header with a short summary of all the transaction in it and some metadata. Finally, the node perfoms the ***PoW - Proof of Work***.

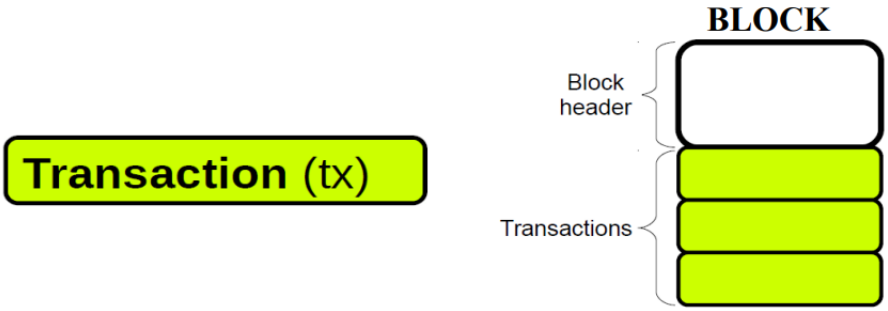
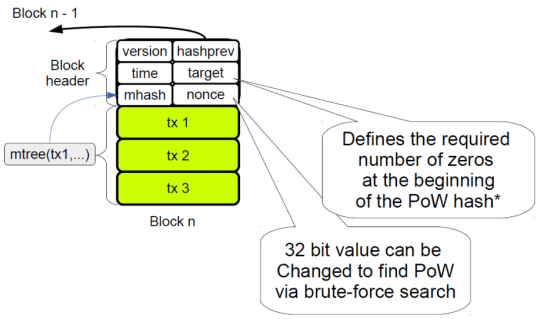


Figure 3.13: Block structure

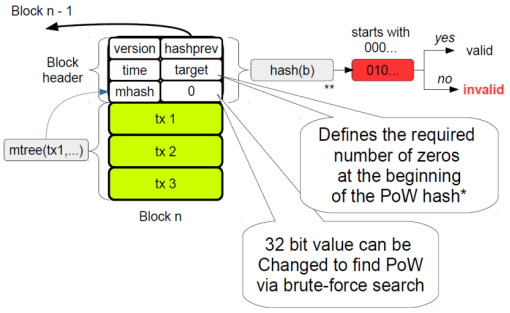
Usually the transaction list is almost 1000x larger thhan the block header. The header contains:

- **version of the protocol**: used for bug fixing and new features
- **time**: local clock timestamp. It's used for tuning the difficulty of PoW. It indicates the time of block creation.
- **mhash**: it's the root of the ***Merkle tree*** built on the transaction. Te leaves of the tree are the block transactions. The merkle tree it's built on demand so it's not explicitly represented in the block. Any change to a transaction rresult in change of both the *mhash* and the hash of the whole block. Having the root of the Merkle tree allows to check the transaction integrity.
- **hashprev**: hash of the previous block, the block over which the miner build the current block. It's a SHA-256 field. There are other two fields called **target** and **nonce** that are realted to the POW (*see later*).

The **mining process** works as follows: set the value of the nonc of the block header to 0, then compute the hash of the whole header and check if the resulting hash is under a certain threshold setted by the **target** field. If the hash of the block header is not below the target, increment the **nonce** (3.14a) but mantains all other information equals to the previous: this allows to obtain a completely different hash. **Keep trying by incrementing the nonce until you find a value under the treshold.** In 3.14b an example of unsuccessfull nonce finding.



(a) Successfull nonce values

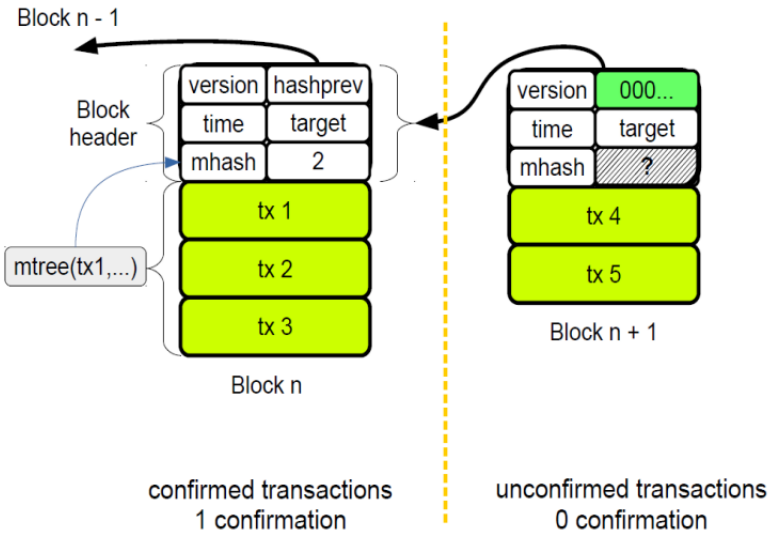


(b) Unsuccessfull nonce values

The difficulty of founding a suitable nonce can be tuned as well as different blocks may have different **target values**. The tuning is performed based on the number of nodes and the total computational power of all the nodes in the network.

Random Node Selection

The selection of the nodes is randomly executed: the key idea is that the nodes are selected in proportion to a resource that it's hard to monopolize. In Bitcoin, this resource is the **computational power** and the selection is done on the basis of the Proof of Work.



The **Nakamoto consensus** works by selecting a leader node at random: the selection is implemented through the PoW and who is able to find the right nonce so to win the PoW, is seleted node and entitled to propose the next block. At least 51The selected node *unilaterally proposes* without contacting other nodes, the next block to be inserted in the ledger: can be viewed as a random leader election on every block. THe block is broadcasted in the network so all nodes cna check the validity and update their blockchains with the new chosen block.

Proof of Work - Formal definition Let's define as:

- d: the difficulty as a positive nuber which is used to adjust the time to execute the proof
- c: the challenge as a given string (*the block header minus the nounce*)
- x: nonce as an unkwnon string The proof of work is a function defined as $F_d(c, x) \rightarrow \{true, false\}$ that satisfy the following properties:
- c, d are fixed
- $F_d(c, x)$ is fast to compute if all the three parameters are known
- Finding x s.t. $F_d(c, x) = true$ is computationally difficult but feasible

The proof of work is **hard** to solve because the output looks like a random 256 bit string where each bit is equally likely to be 0 or 1, independently of the other bits. No better way of finding the correct output than trying **brute force**: the probability p that the block hash falls below the target threshold T is: $p = \frac{T+1}{2^{256}}$ and the average numbers of trials required to find ablock hash below the treshold 1/p.

Mining reward The mining activity is rewarded, incentivizing miners to be honest. Two main mechanism are implemented:

- **Block reward**: a payment to the miner in exchange for the service of creating a block. Bitcoins mints new coins whe a new block is mined. It's the only way to create new bitcoins.
- **Transaction fee**Unknown Node :: textDirective each transaction is the block, they take the difference between transaction input and output. It was voluntarly inserted to obtain a good quality of service from the miners.

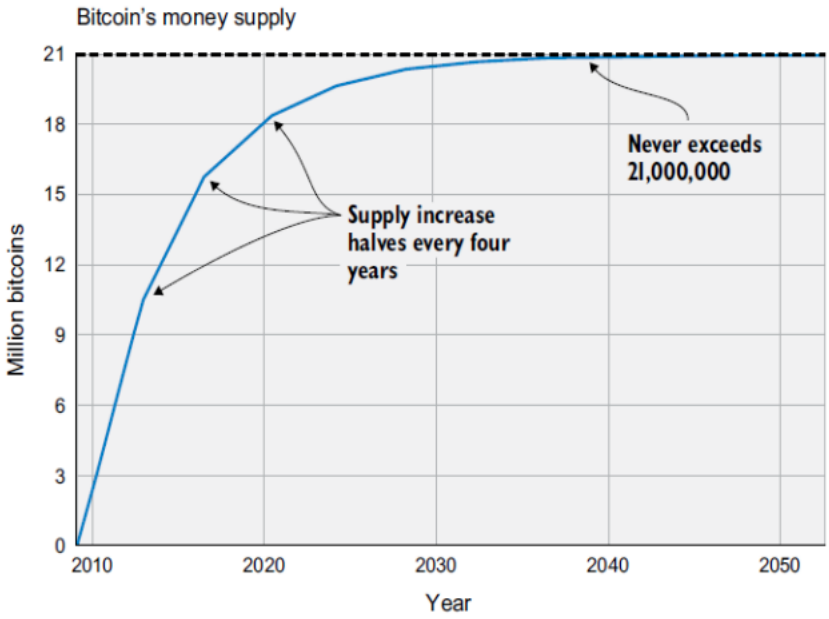


Figure 3.15: Bitcoin limited supply over the years

The **coinbase** is the first transaction in each block: it encodes the trasnfer of reward plus the transaction fees to the miner and does not consume previous unspent contained in the blockchain but uses a **single, dummy input** (*without reference to the previous transaction*) while the output is the sum of the rewards and fees that are given to the miner in one or more addresses. The rewards dynamicaly varies in time: halve the mining reward after all 210.000 blocks (*every 4 year*). Each period is defined as *"era"*. Those rewards are the only way to generate new bitcoin: they exists in a limited supply in order of 21 millions.

Due to the arise of the miners and mining pool, it's not possible to compete nowadays with miners: they use specialized hardware.

Mining difficult The mining difficut it's **variable**: the frequency with which miners mine a new block depends on the computational power inside the overal network. The goal, fixed by Nakamoto, was to have a block mined *every 10 minutes*: this target it's not really fixed but it's adhusted by the protocol to acheive the goal. The goal depends also on the network condition to propagate the block to all/most of all nodes: the idea was to allows time to propagate across the whole network before the next blockg ets mined. This allows to avoid incosistency between ledger's node (*differently from Eth that mines a block every 20s*). The rationale behind the 10 minutes frequency is given by the followign example: if blocks are mined too frequently, iners can **build competing chains** but only one of this will become the longest so some miners will last energy building ona chain that will be left out.

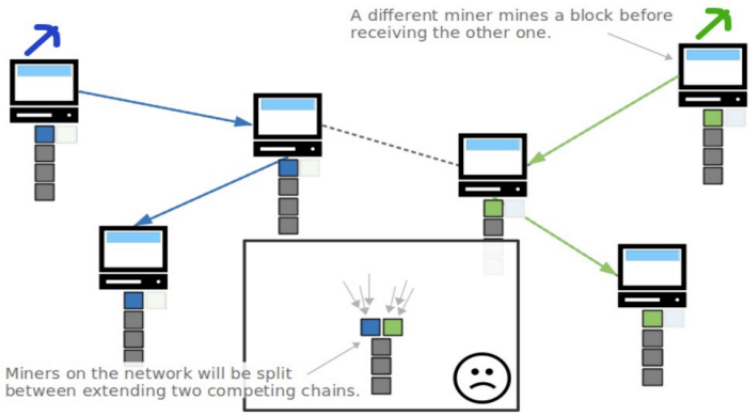


Figure 3.16: Competing chains

The ideal scenario is represented by all miners concentrating the network's mining power on extending the same chain. A shorter block time have advantages like *faster confirmation* and *less payout variance for miners* as Ethereum does by implementing more forks, introduction of **ommers** (*is a term used to refer to a type of block that is a direct descendant of the parent block, but was not included in the main blockchain*) and a more complex rewarding system.

To tune the difficulty, the target can be changed from one block to another: here two different target, with the longest sequence of 0's the more difficult target (3.17). Increase or decrease the number of starting zeros bits in the target, according to the ratio *expected time/actual time* allows to tune the difficulty. If blocks are mined faster than every 10 minutes, the target will adjust *downwards to make it more diificult to get below the target.*

0	03 Jan 2009, 18:15:05	00000000ffff0000000000000000000000000000000000000000000000000000
727,776	17 Mar 2022, 17:40:08	0000000000000000000a40c0000000000000000000000000000000000000000

Figure 3.17: Two different target complexities based on starting zeros

Every node can determine the checkpoint of mining based on the time fields in each block: 2 weeks is an approximation of the time required to mine 2016 block (*as* $2016 = 14 * 24 * 6$) that is the **number of blocks mined in two weeks if a block is found every 10 minutes**. Compute th ratio between the actual time between blocks and the expected amount of time and if it's ¡ 1 it indicates that the mining of new blocks is too fast. This allows the nodes to understand if the *10 minutes goal* is satisfied or not.

Every nodes execute the same algorithms so they **adjust** the complexity in the same way as the other nodes: every node executes exactly the same algorithm so all nodes adopt the longest chain of blocks as their blockchain. They will end up calculating the same target as every node else thus all nodes end up sharing the same *current target* for the same block.

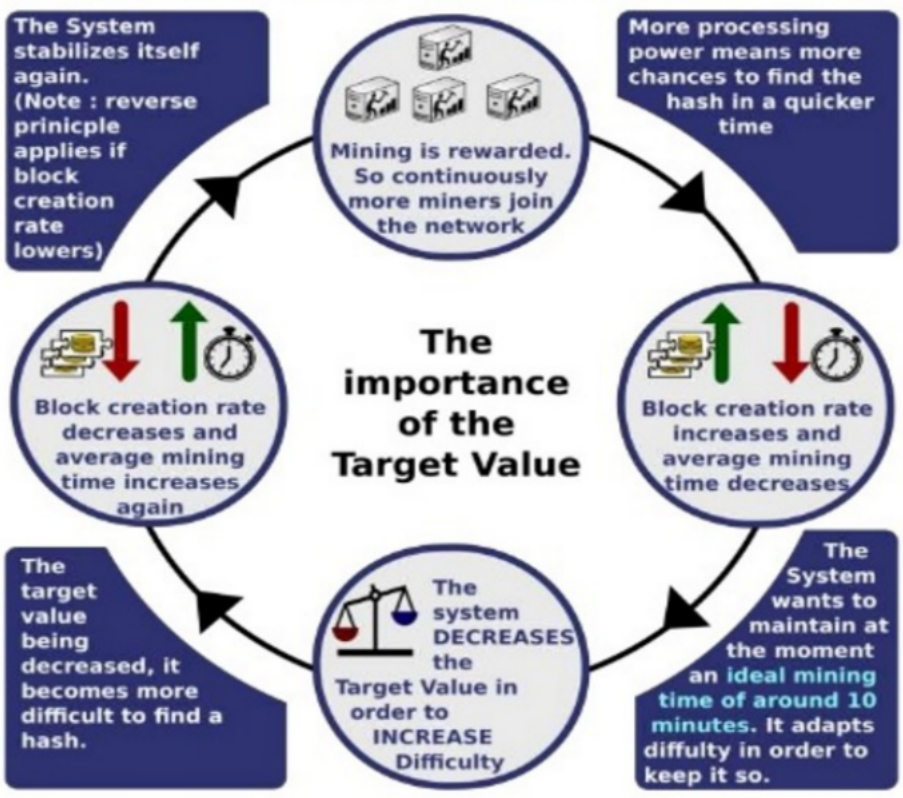


Figure 3.18: undefined undefined

BTC Blockchain overview The **Bitcoin blockchain** consists of a liner list of blocks where each block is composed by *a block header* and a *list of transaction* structured as already seen: blocks are linked each other through **hash pointers**.

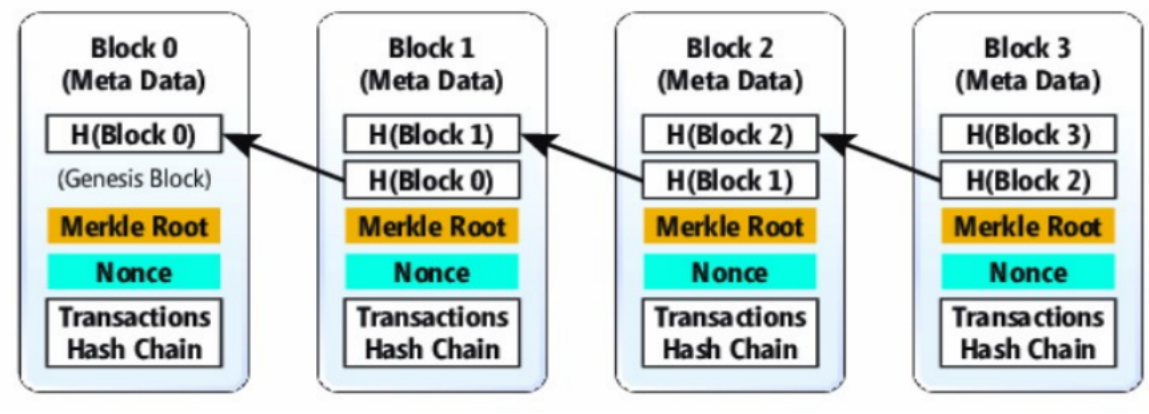


Figure 3.19: Blocks information chaining structure

Blocks are **uniquely identified** by:

- ***Block hash***: it's computed when the block is received and not included in the block. It facilitate the block indexing and retrieval for future checks on transactions.
- ***Block height***: number of blocks proceding it in the blockchain, starting from the *genesis block*.

The **tamper freeness** is guarantee by a combination of the *nonce* and the *hash pointers*. Image an attacker changing a transaction in a block: the **root of the Merkle tree** changes so also the block header change but this invalidates the *nonce* and the *PoW* must be re-executed to recompute the right *nonce* for the new block. In the next block, the *hash pointer* to the previous block changes as well because the *nonce* of the next block is no more valid and the ***PoW* has to be re-executed for the next block**. This process is repeated for the entire chain so a potential attacker would have to recompute the PoW for the **entire chain**, using an enormous computing power.

A **block** represent the **unit of work for miners**: the base building block isn't a single transaction because, compared to blocks, those have a *shorter hash chain* that allows to fasten the verification process. Also, mining a single transaction would require more overall mining work and less efficient communication for transferring a single transaction instead of an entire block.

Temporary forks It's possible that at a given time, we have two miners *"winning at the same time"*: this is possible if two miners validate and broadcast a block simultaneously. This scenario result in a **fork** of the blockchain that now have two different branches. The state of the blockchain is seen by the network consists of two braces both originating from the same *parent block*: both branches are *legitimate* because are originated from the correct execution of mining algorithm by honest nodes.

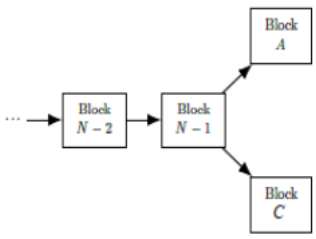


Figure 3.20: Temporary fork

This create the problem of **reconciliate the two versions** of the status of the blockchain: in the Bitcoin blockchain, the applied solution is known as ***Nakamoto rule*** in which "*the longest fork wins*". Basically, 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 return to the pool of

"not yet approved" transactions. This is the key idea behind the **6 confirmation rule**: Bitcoin approves a transaction finaly once there are **at least five following blocks in the chain**. So a transaction is not considered confirmed until at least 5 blocks follow it in the *longest fork*: 6 it's the default value but can be choosen by the client itself. In figure 3.21 the sketch of the algorithm.

```
Receive block b
For this node the current head is block b_max at height h_max
Connect block b in the tree as child of its parent p at height
        h_b = h_p + 1
if h_b > hmax then
        h_max = h_b
        b_max = b
compute UTXO for the path leading to b_max
cleanup memory pool
end if

UTXO: Unspent Transaction Cache
```

Figure 3.21: Nakamoto rule pseudo-code

The previous rules,combined with randomness introduced in the mining algorithm and block propagation **ensure that the probability to obtain two parallel chains of equal height is the lowest**.

Nakamoto consensus The theorem states that: "*forks are eventually resolved and all nodes eventually agree on which is the longest blockchain. The system therefore guarantees eventual consistency.*"
The proof is here sketched:

- in order for the fork to continue to exist, **pairs of blocks need to be found in close succession**, extending distinct branches
- otherwise the nodes on the shorter branch would switch to the longer one
- the probability of branches being extended almost simultaneously **decreases exponentially with the length of the fork**
- hence there will eventually be a time when **only one branch is being extended**, becoming the longest branch.

The overall scenario is pictured in 3.22.

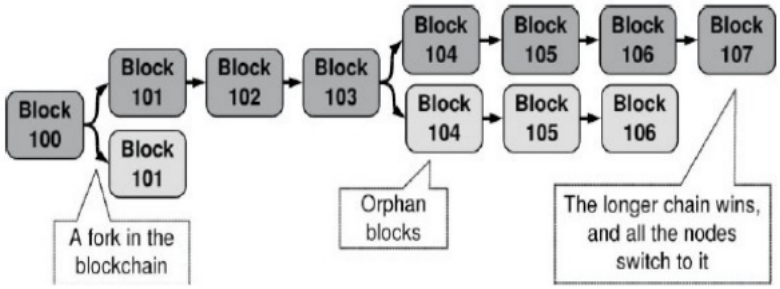


Figure 3.22: Branching scenario

So instead of a blockchain, a **tree of blocks** is obtained: the path in the tree corresponding to the longest chain is the blockchain while dead path contains the **orphan blocks**.

3.4 Attacks: selfish mining and malleability

To discuss about attacks, we first describe the simplest scenario: *no blocks are mined at the same time* so no attacks are possible to be carried out. A new block `B` is mined by a miner: this block is shared by other nodes so when other nodes receives B, they add it to their blockchain. Every miner will start trying to mine a new block *on top of* `B`. Another scenario is when we have to concurrent node, mined at the same time, that form a fork in the blockchain in *two directions*: both blocks are correct because are mined by following the protocol by trusted nodes. For this scenario, as discussed in L12 - 5-04-2023, the solution is to use the **6 confirmation rule** or **Nakamoto rule**: all the transactions in the dead branch are not valid because are not recognized officially by all nodes as the longest and thus the valid chain.

This approach follows the **longest chain rule** that incentivize the miners to focus only on the longest chain. This implies to reorganize the chain: this lead to the possibility of *replacing blocks* that can be exploited for attacks.

Here an overview of attacks in *Bitcoin Network* with some countermeasures:

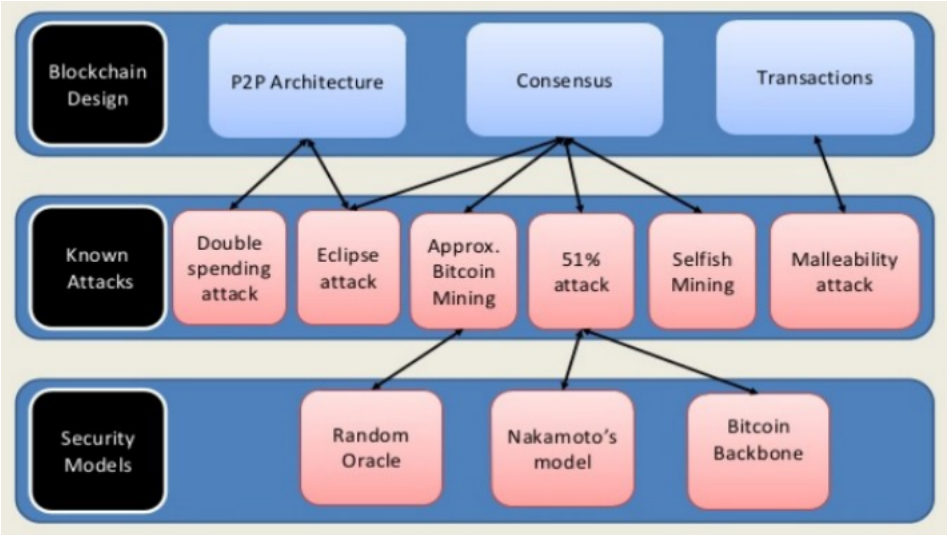


Figure 3.23: Taxonomy of BTC blockchain's attacks

The malleability attack was based on an problem regarding the transaction algorithm that allows nodes to violate the trustness and modify the transaction information.

1. Double Spending Attack

Basically, it tries to spend the *same bitcoin twice* by sending them to two different recipient at the same time.

Type 1: First, simple strategy to carry out this type of attack is to send two transaction to the network: both transaction go in the *MemPool* of the miners so only one will be inserted in the next block from a *honest miner* while the second will be considered *invalid* because recognizes that the sender is trying to spend the same UTXO in two different transactions.

Type 2 The scenario get more complex if *those two transaction are validated by* ***different*** *miners*: this is the scenario where the blockchain forks, realizing the case already mentioned, solved by the *Nakamoto rule.*

Type 3 Another scenario is when the sender is also a **malicious miner** so he can validates a block and includes both transactions in the same block. Other nodes check the validity of the block and simply reject it: the attack does not succeed so mining effort is wasted.

Type 4 From the previous case, we can derivate another method, pictured in 3.24: the sender is still a **malicious miner** and tries to perform a double spending attack by *"reversing"* the longest chain by **performing mining in stealth mode** so do not broadcast/propagate it's chain to the rest of the nodes immediately, delaying the propagation to the entire network.

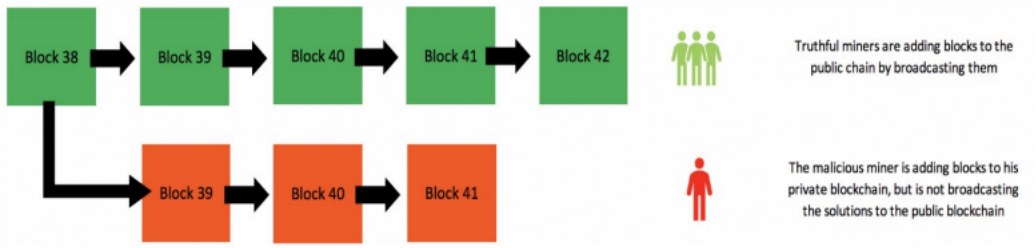


Figure 3.24: Type 4 scenario

Type 5 Another type of method involves using an *honest branch* mined by a *honest miner*: the sender is still an **untrusted miner** so in his *stealth blockchain* he exclude his own transaction (*that was in the green, legitimate branch, pictured in 3.25*), inserting the *orange* blocks.

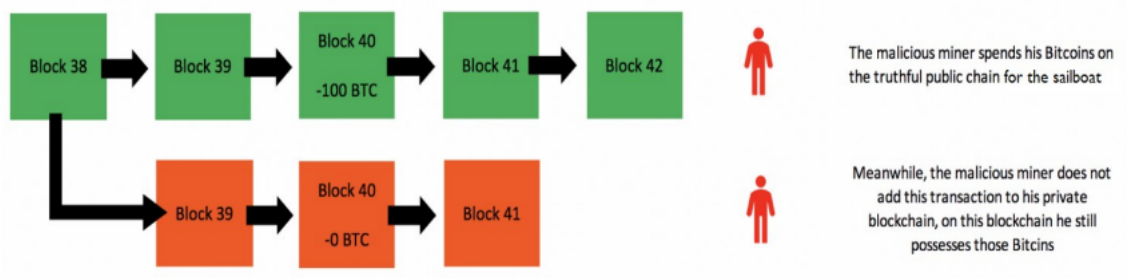


Figure 3.25: Type 5 scenario

Type 6 (51% attack) Another method is that the malicious sender creates a longer chain, broadcasting his version of the chain to the rest of the netwrok. It will be acceptd by the miners due to the **longest chain rule**: the malicious miner needs more hashing power than the rest of the miners, known as *51%* of the hashing power. So this allows to add blocks to his version of the blockchain faster and eventually build a longer chain, as shown in 3.26

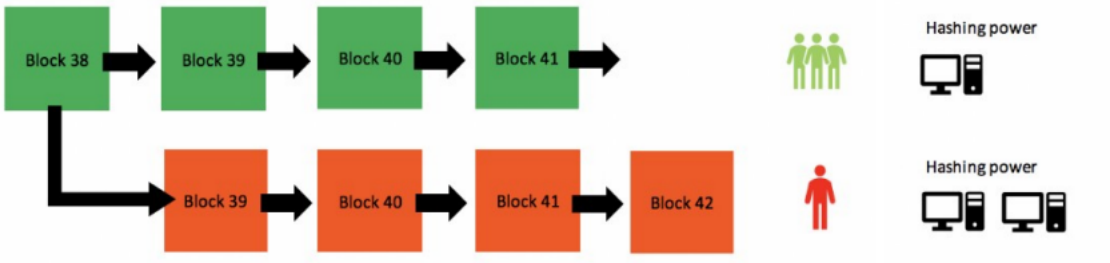


Figure 3.26: Type 6 (51% attack)

As soon as the corrupted miner creates a longer blockchain, it ***suddenly broadcasts*** its blockchain to the rest of the network that must accepts the "*corrupted*" version of the blockchain as enforced by the **longest chain rule**. The chain is longer than the one they were working on and they switch to this chain. If the attack succeeds, the old chain is abandoned, and transaction in that chain are no more valid so the malicious sendr is able to spend again the same amount, already spended ion the abandoned branch.

An example of 51% attack was carried out by the mining group called *Ghash.io*. It came close to reaching 50% in 2014 but even with 50% this doesn't necessarily mean that it will perform an attack. The case of Ghash shows that is more remunerative to mining respect to perform an attack that lead to a price crash.

Transaction Malleability Attack

It's based on the fact that you can modify the *transaction hash (**TXID**)* **without modifying** the signature of the transaction. It's noticeable to remember that the input of a transaction is signed by the private key of the owner. All the nodes receiving the transaction verify that the sender is really the owner of the transaction. The transaction identifier `TXID` is the SHA-256 of all the fields in the transaction data: so it depends from the *unlocking script (signature script)* but a node can **change the unlocing script** in such a way that:

- the transaction has the same effect
- the signature is still valid
- the `TXID` **changes** This implies that the signature is still valid despite the transaction ID changes.

Alice issues a BTC payment to Bob, and signs the transaction $TXID_1$: Bob alters Alice's transaction signature, before $TXID_1$ is confirmed, this results in a new transaction identifier, $TXID_2$, on the same content. $TXID_2$ gets confirmed on the blockchain before $TXID_1$ so the latter will not be confirmed, because it appears as a double spending. Alice does not see her transaction confirmed amd Bob could then defraud Alice by asking her to issue a new payment. Alice belives this, because she see another transaction confirmed, not her transaction: a double spending attack performed by the receiver of the transaction allows Bob receives twice the intended amount.

The solution to avoid the **malleability attack** was to change the structure of bitcoin transaction data, kwnown as **Segregated Witness (SegWit)**. The key idea was to separate all the malleable information into a separate *witness data* and compute the TXID without the *unlocking script* so the transaction identifier will never be able to change again. Here is pictured the process before the fork, called *SegWit*, and after:

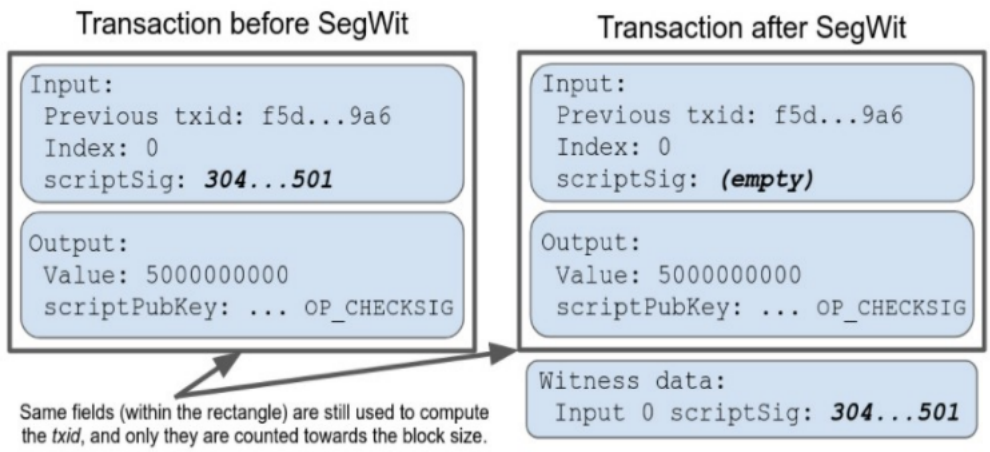


Figure 3.27: Pre SegWit vs After SegWit

Denial of Service

As an example, imagine that a miner, say Alice, really dislikes some other user, say Bob, and want to denies service to him. Alice decides that she will not include any transactions from Bob's addresses in any block that she proposes for the block chain but if Bob's transaction is not inserted into the next block that Alice proposes, it remains in the pool of *unconfirmed transactions* of all the miners Bob will just wait until that an honest node proposes a new block so his transaction will get into that block, so being confirmed by the network thus the block will be inserted in the blockchain.

3.4.1 Bitcoin ecosystem

A bitcoin client can implement different features, based on its specific type:

- **Reference Ciient (Bitcoin Core)**:
- **Full Block Chain Node**: contains the full blockchain database and network routing but not partecipate to the mining of blocks.

- **Solo Miner**: contains mining function with a full copy of the blockchain and the bitcoin P2P network routing protocol.
- **Lightweight Wallet (SPV)**: contains a wallet and a network node on the bitcoin P2P protocol, without having the full blockchain.

Miners

Basically execute a continue HASH computation, with the core cycle composed by:

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

Those activity lead to the design of specific hardware for mining, impliying an **increasing difficulty** over the year to be able to mine a new block. This was solved by Ethereum with PoS. The first generation of mining was **CPU-based**: was based on the sequential search of right nonces or a few core parallization, compuitng a SHA-256 directly in software, without specialzied hardware.
The second generation was based on **GPUs** designed for high-performance graphics: this allowed to use an high degree of parallelism, computing different hashes with different nounces. This allowed the miners to achiee high throughput. This approach lead to an under-use of GPU, consuming a lot of power.
The hird generation was **FPGA-based**: they were programmed in Verilog and ensure performances as close as possible close to the custom hardware. This lead to design high performance GPUs, allowing excellent performance on *bitwise* operations.
The fourth generation was based on **ASIC Application Specific Integrated Circuits**: is an integrated circuit chip designed for a specific purpose. An ASIC miner is a computerized device that uses ASICs for the sole purpose of mining digital currency. Generally, each ASIC miner is constructed to mine a specific digital currency. So, a Bitcoin ASIC miner can mine only bitcoin. One way to think about bitcoin ASICs is as specialized bitcoin mining computers optimized to solve the mining algorithm by implementing a fast computation of *SHA-256*.

Two main approaches to mining:

- **Solo mining**: a single miner try to solve the computation problem. This scenario is described as a Poisson process
- **Mining pools**: miners group together to create **mining pools** that allows to reduce the variance of their income.

I the first case, the mining is a very risky task even if the reward is high: there is an high probability of spending a lot for mining hardware and electricity withouut obtainin a reward for a long time.

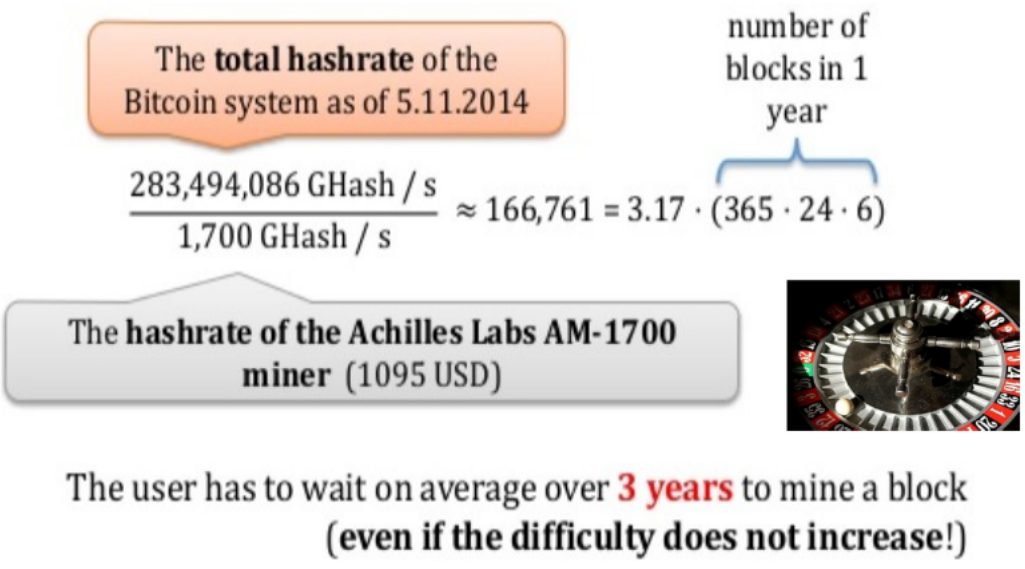


Figure 3.28: Hashrate computation example

The bitcoin mining follows the Poisson distribution, having a large standard deviation.
For the second case, we have **mining pools** that ma operate in two different ways:

1. throuht a **centralized mining operator** so the miners should trust the pool manager
2. in a P2P way, using a private blockchain to manage the pool

For the **centralized** one, the pool manager send the blocks to all the miners, distributes revenues to members based on the work they have performed but the manager must be trusted by everyone. This implies also that nodes in the pool need to produce a measure of their work to be rewarded proportionally for his work, even if they are not able to solve the given problem and find the nounce.

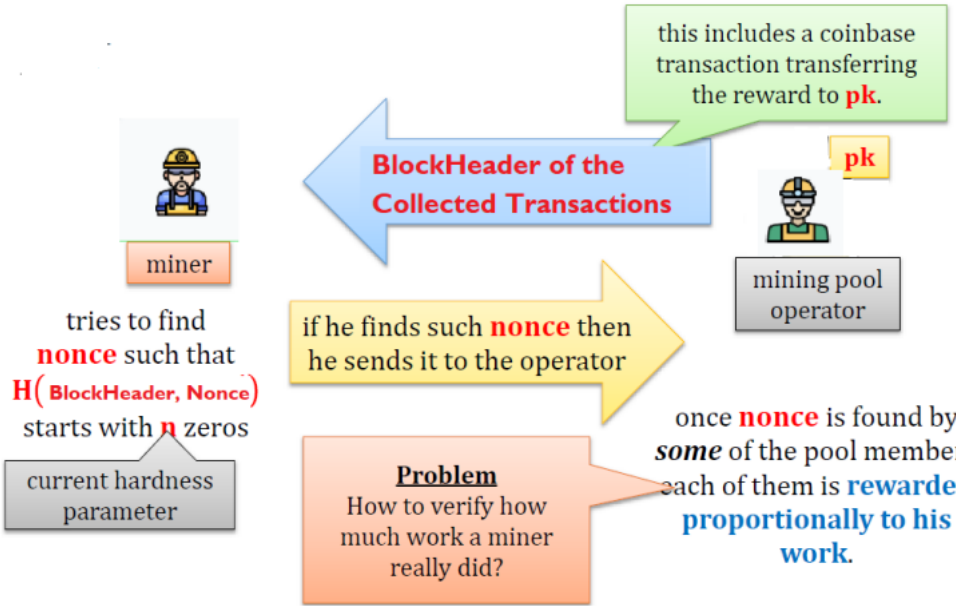


Figure 3.29: Centralized mining operation workflow

This approach poses challenges like *how does the pool mananger known how much work the node of the pool are performing?* The proposed solution is called **Mning shares**: it's a proof of the work done by a miner, sent to the mining pool cordinator. It's based on an hash called *near-valid blocks* or *near optimal hashes*: the hasesh contains fewer zeroes than those required by the current difficulty. This does not entirely shows that hard work have been done but probabilistically prove how much work has been done.

Payout In the **pay per share** model, the miner send to the mining pool operator the *partial solution* and receive a *reward* based on fair share of work executed and demonstrated. The instant payout guarenteed to the miner presenting a share: this guarantees a reward without waiting for the pool to find a block but no bonus is paid to the miner who actually mines the block. The miners are payed from pool's existing balance so the risk is at the pool's operator but he is also protected by the participant fees. The main disadvantages are that workers do not have valid incentives to send valid blocks to the manager so they can discard valid blocks but still be paid the same reward.

In the **pay proportional** model, the amount of payment depends on the whether or not the miner actually found a valid block. Every time a valid block is found, the rewards from that block are distributed to the members proprtional to how much work they actually did. This lower the risk for the pool manager because he payout only when a valid block is found.

Decentralized mining pool

The key idea is that no operators is required to verify each miner's contribution to the pool: this is possible by building a **separate, private chain** that include also *"weak blocks"*, mined with lower difficulty. It's necessary to store in the private chain the transaction rewarding for mining weak blocks, until a valid block is found, then distribute the reawrd through a side blockchain which is then merged to the main chain, by a process kwnon as **merge mining**.

So miners in the pool create a private blockchain, called **sharechain**: the difficulty of mining a block can be $n^1 << n$ where n^1 is choosen such that a new block appears every 30 seconds. The block is built on top of the last block of the *public blockchain* so each block of the private blockchain correspond to a *share of work* done by one of the miners: this allows to avoid a central operator to check, by making the process transparent.

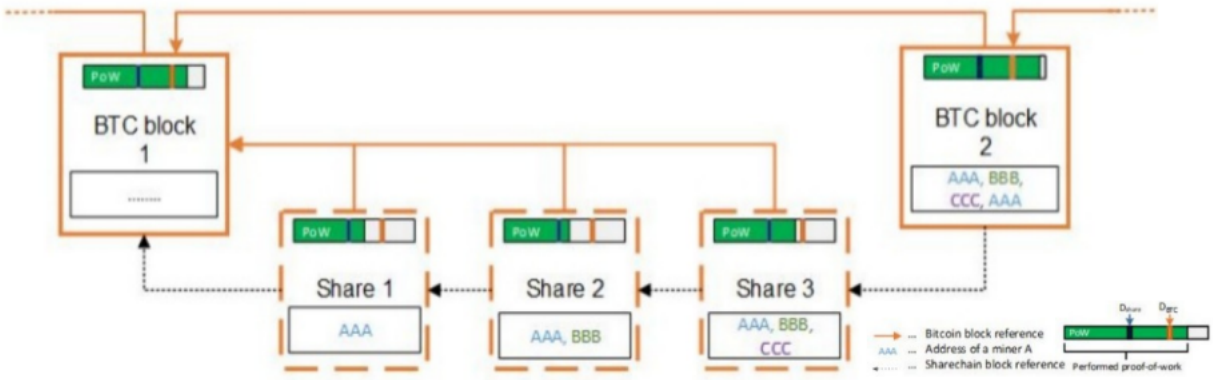


Figure 3.30: Sharechain creation

The first block of the shared chain inclue the payment to `AAA`, the first miner generating a share. The second block includes a payment both to `AAA` and to `BBB`, the second miner solving the *reduced* ($n^1 << n$) PoW. The process iterates. When a miner solves the PoW, the last block is linked to the "*real*" public blockchain so no miner can cheat avoiding to insert in the next blocks a payment all the payment to the previous ones. This is the blockchain as seen by the rest of the network after a valid block is found in the mining pool:

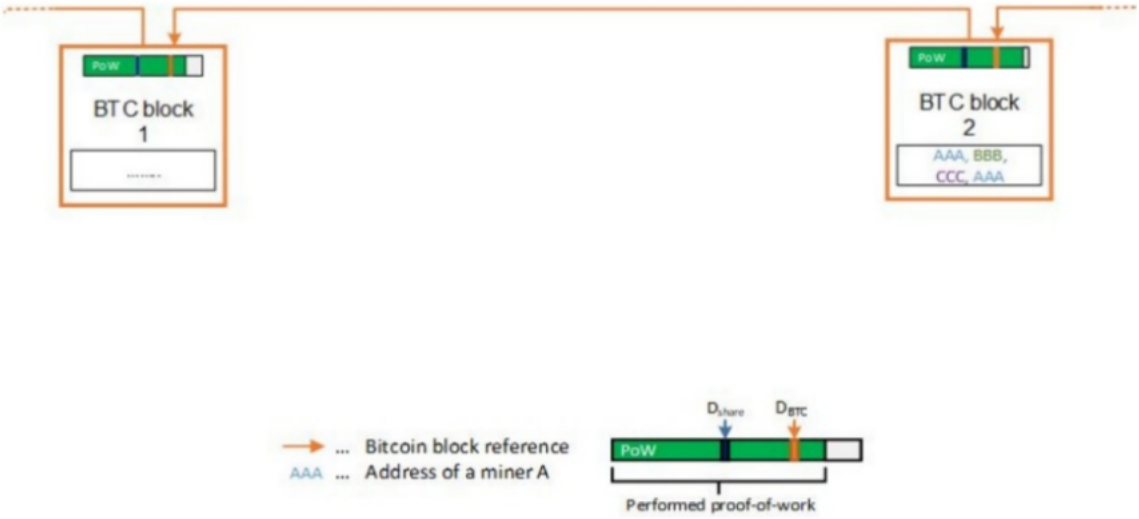


Figure 3.31: Blockchain after merging with the real one

Chapter 4

Ethereum

Ethereum is a blockchain platform for building decentralized applications: *application code and state* is stored on a blockchain and transactions can cause code execution and update state, emit events, and write logs. Even frontend web interfaces can respond to events and read logs. It's the most used platform for NFTs and ICOs (Initial Coin Offers). ICOs are based on ERC-20 token contract. It's useful in a scenario where a company seeking to raise money to create a new app, or service can launch an ICO as a way to raise funds. The interested investors receive a new cryptocurrency token issued by the company. This token may have some utility related to the product or service that the company is offering or represent a stake.

Ethereum allows to implemented **smart contracts** that are *decentralized, replicated, processed* on all the nodes on the network, without a central coordinator. The consensus mechanisms assure that all the nodes agree on the **results of the execution** and update the state in the same way so each node changes its own version of the ledger with the result of the smart contract evaluation. This is implemented by a a **distributed state machine** that define how change the **global state**, which is the status of all the smart contracts. A single transactions allows to change the global state.

4.0.1 Overview

As Bitcoin, Ethereum is based on a ***transaction-based deterministic state machine***: a virtual machine that applies changes to a global, replicated state. The state change is triggered by a transaction but Ethereum allows anyone to create its own **state transition functions** without imposing any script. Differently from Bitcoin, Ehereym uses **accounts** that keep track of balance: the global shared state of Ethereum is stored in a **multitude of accounts**, from small objects interacting with one another through a message-passing paradigm.

Compared to BTC, Ethereum is a a **public** and **permissionless** blockchain where the addresses are generated by *keys* and the transactions are signed through a *digital signatures*: in Ethereum blocks contain data and smart contracts, in Bitcoin blocks contain data and scripts. The **consensus algorithm** was Proof of Work until 2022, when Ethereum introduced Proof of Stake. As Bitcoin, Ethereum is deployed on a P2P Network: Kademlia i'ts used at P2P levelto discover peers.

Accounts overview

Each account has an identifier of 20 byte plus a state indicator. There are two maint ypes of accounts:

- **Externally owned (EOA)**Unknown Node :: textDirective by private keys so they don't have any code associated with them.
- **Contract accounts (CA)**: have code associated with them thus are controlled by the associated code. The first type of accounts usually contains informations like *address, Ether balance, nonce (total transactions emitted from that account)* and are able to send transaction to transfer Ether or send transaction to trigger a smart contract. Differently from BTC, a transaction it's not based on UTXO but it's **account-based** (*as suggested by the balance and nonce*).

In case of **Externally Owned Account (EOA)**, the transaction format must include *Signature, Receiver address and the Amount.* In case of **Contract accounts (CA)**, the account contains:

- *contract code*

- persistent storage for contract variables
- *balance*: like EOAs, contract accounts have an Ether balance and can receive/transfer Ether
- *nonce*: number of messages sent from this account

Smart Contract A smart contract is a computer program in which the code cannot change after being deployed on the blockchain because it's executed in parallel by all the *full nodes* of the network. The code must define a **deterministic computation** because the output of a smart contract must be the same for every node: to execute a contract, an *execution context* myst be provided. The context include data taken from the transaction that has activated the contract, the internal storage of the contract and information from block headers of the blockchain. We can distinguish three phases in the lifecycle of a smart contract:

1. **Creation**: To deploy a smart contract,the developer need to have an **EOA**: the transaction to create a contract involve using the information alredy defined for EOA transaction but with the *Receiver address* empty and with an extra field called `data` that contains the code of the smart contract.
2. **Interaction**: The *EOA* calls a method of a smart contract through a transaction, enabling the execution and passing the required arguments. This phase uses a transaction with the *Receiver address* setted with the address of the contract and with the `data` field containing the method to call and the parameter to pass. Smart contracts are triggered by transactions from EOA or from messages from other smart contracts which call functions inside a contract, specifying the address of the contract and the parameters of the function. The transaction may even contains Ether to transfer to the smart contract. Contract account **cannot initiate a transaction by themselves**: when activated may call other contracts, but not themselves. This can lead to build complex execution paths. Upon reception of a transaction/message, the **Ethereum Virtual Machine (EVM**) is executed: it can perform different actions like *computation, write internal storage, send messages to another contract or create a new contract*. The process is sketched in 4.1.
3. **Destruction**: this phase is triggered by a transaction with the *Receiver address* setted as the contract address and the `data` field containing the name of a method that calls the *self-destruction operation*.

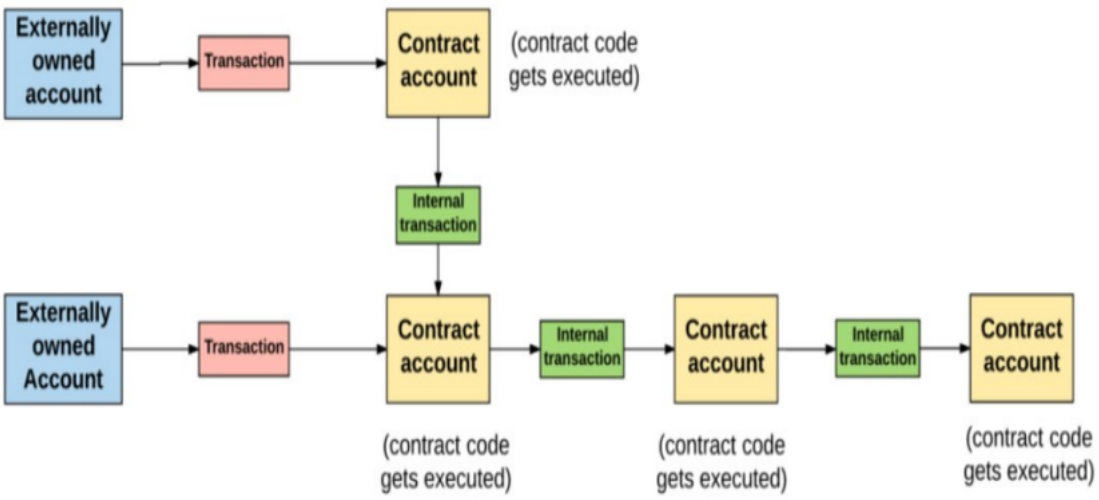


Figure 4.1: Interaction process of a smart contract

So, overall, **EOAs** are a bridge between the external world to the internal state of Ethereum: any action is always set in motion by transactions fired from externally controlled accounts.

Accounts and transactions insights

An account, as mentioned, contains:

- `nonce`: number of transactions sent/contracts created

- `balance`: owned *gwei (see Gas section)*
- `storageRoot`: i'ts the digest of the Ethereum's state as a hash of the root node of a Merkle Patricia trie.
- `codeHash:` hash of the code of contract account account. The `hash("")` function is used for external accounts.

Focusing on the **nonce**, the yellow paper states that is "*a scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account*" so it's an *attribute of the address that originate the transaction* and not of the transaction itself. The nonce allows also to records the order of the transactions, protecting against **transaction duplication** and **reply attacks** alongisde **double spending** from the same address. It's noticeable to remark that there are two types of *nonces*:

- **Transaction nonce**: not as Bitcoin nonce
- **Block nonce**: used by PoW, like Bitcoin's nonce

In case of **reply attacks**, because Ethereum stores account balances, the attacker takes any existing transaction and resend it on the network more and more time. All these replayed transactions try to withdraw funds from the same account and are considered valid because are signed with by the account's owner. Thus if the account has enough Ethers, the attack is successfully. Ethereum prevents this with the using the **account/transaction nounce**: suppose that the `nonce` of the original transaction is 22,after receiving the original one, the network does not accept another transaction from the same account with nonce setted as 22 and this prevent the attacker to change the nonce because would invalidate the signaure of the transaction. Each time an account sends a transaction the nonce is increased by 1 so it's used to enfroce the rules to consider a **transaction valid**. The transactions **must be in order** so a transaction with nonce 1 cannot be mined before one with nonce of 0. Also, transaction **cannot be skipped**: if a transaction has nonce 2 cannot be mined if the miner has not already sent transactions with nonce of 1 and 0.

To **propagate a transaction**, Ethereum uses a *flooding approach*: the propagation starts with the node creating a *signed transaction* so the transaction is validated by each node and then it's transmitted to all other nodes that are directly connected to the originating node. On average, Ethereum node mantains connection to at least **13 other nodes**: each neighbor validate the transaction as soon as it receive it. If the received transaction is valid, they store a copy and propagate it to all their neighbords (*except the one it came from*).

The process of **transaction mining** modify the global state of Ethereum: the balances of the involving accounts are modified and all contracts are activated by transactions/messages that are executed. The state of the accounts are updated and stored in a **Patricia Merkle tree** so a new `stateRoot` is stored in the block header and the **Receipt Root** it's updated to contains the generated events.

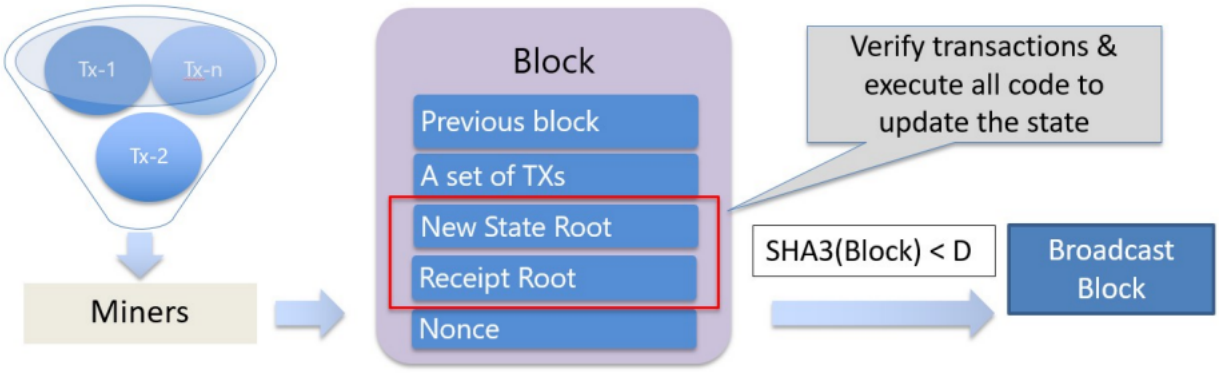


Figure 4.2: Transaction mining and propagation

The winning miner will publish the block to the rest of the network,the other nodes execute the contracts and check the result. If the result is valid, they add the block to their blockchains: the entire network agrees on the *current balance, storage state, contract code, etc*, of every single account.

Ethereum State Machine

The state of the Ethereum network comprehends the **state of all its accounts**. A simplified version of an account state the *account address, balance and code and state of the variables for contract accounts.*

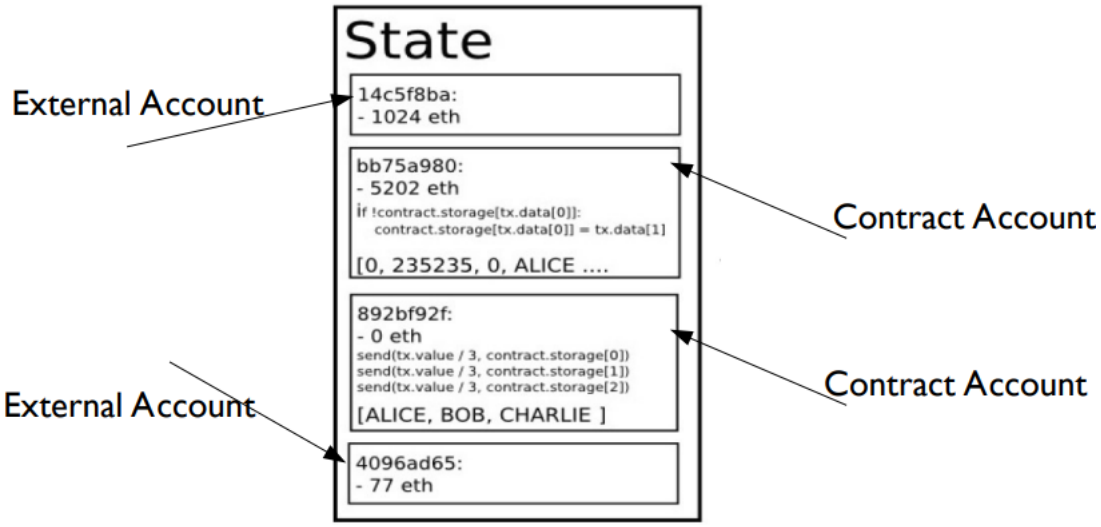


Figure 4.3: State of an account

Here a simplified version of **state transition fired by a transaction**:

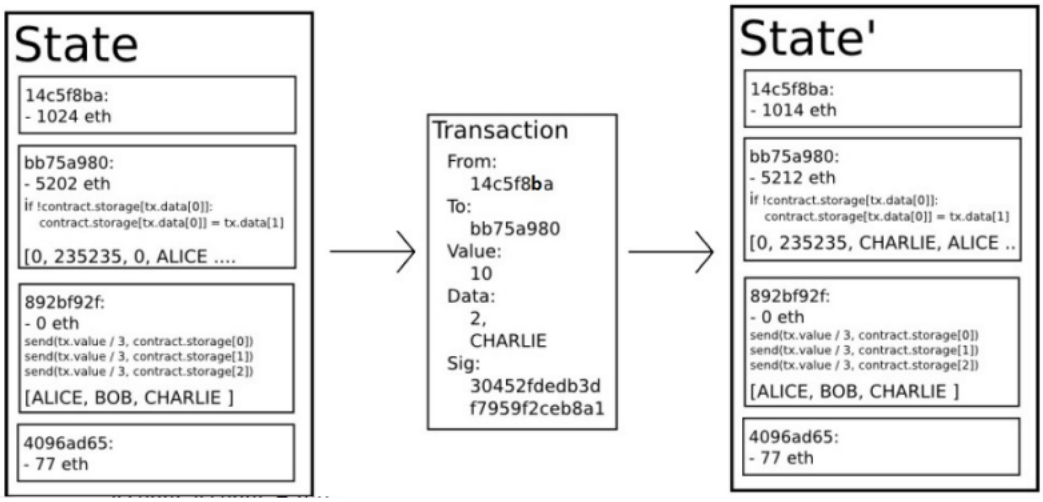


Figure 4.4: Account's state transition after a transaction occurred

4.0.2 Gas

Every node in the network evaluates all transactions and store all the *contract state*: it's not possible to tell, just by looking at the program, whether it will take forever or not to execute. The program must be executed to verify this: due to Turing completeness, this may happen in Ethereum. To avoid the scenario in which a malicious user try to execute a never stopping contract-code, leading to a DoS (*because the EVM is a* ***single-threaded machine without scheduler***), Ethereum introduced the concept of ***gas***.

The key idea is to *pay for contract execution*, giving payed *gas* to your smart contract: this make attacks expensive and allows the EVM to stop the execution when the contract goes out of gas. Each computational step has a fixed *gas fee*, as the *storage* that require also gas fees to perform contract actions. The *EVM* implement a **quasi-Turing-complete machine**: can run any program but only if the program has paied enough gas: at any time, there is a defined *computational limit* given by the amount of gas of the contract.

Gas price The explained model require to buy gas to run a smart contract, so purchasing a distributed, trustless computational power. The gas price in Ether is determined by the *caller*: the price is based on the amount of Ether you are willing to spend per gas unit thus low prices means low priority. The gas prace is also measured in *"gwei (giga wei)"* where one *gwei* correspond to 1 million *wei*. The unit of gas spent of each instruction is fixed:

- **adding two numbers** : 3 gas
- **computing a Keccak-256 hash**: 30 gas + 6 gas for each 256 bits of data being hashed
- **sending a transaction**: 21,000 gas

Both **gas price** and **gas limit** are set for each transaction: the *gas limit* is the maximum amount of gas the sender is willing to pay so the fees are computed as $fees = gas_{price} * gas_{limit}$ which determine the max amount of *wei* the sender is willing to pay for the transaction. The fees are **rewards for miners** for the effort to run computations and validate transactions: the higher the gas price, the more likely miners will select the transaction.

1. if `gas_limit * gas_price > balance` then **halt**
2. deduct `gas_limit * gas_price` from `balance`
3. set `gas = gas_limit`
4. run code deducting from gas the amount required to run code
5. after termination return remaining gas to balance

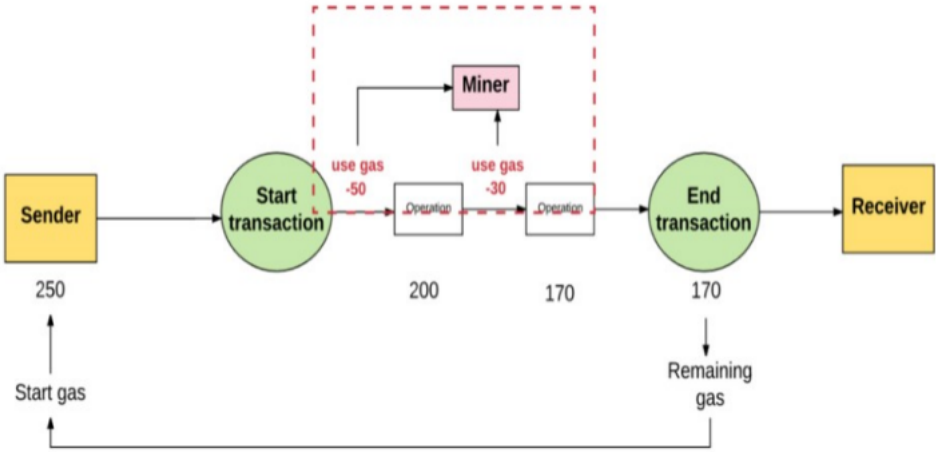


Figure 4.5: Gas workflow process

If the sender does not provide the necessary gas to execute the transaction, the transaction is *out of gas* and is considered **invalid**: the transaction aborts and the state reverts to previous state. Also `gas_limit * gas_price` is still deducted from balance because the node has spent the effort to run the calculations before running out of gas.

The **gas limit** is not considered for internal transaction because is determined by the external creator of the original transction, by some EOA. Surely must be enough to carry out all the transactions, including any sub-executions that occur as a result of that transaction, such as contract-to-contract messages. If, in the chain of transactions and messages, a particular message execution runs *out of gas*, that message's execution will revert, along with any subsequent messages triggered by the execution but the parent execution may not need to revert entirely.

Transaction format The transaction format, sketched entirely in the following image, is serialized using the *RLP - Recursive Lenght Prefix* encoding scheme.

The fields are:

- **TO**: 20-byte address of EOA or contract. There is no validation of the field so if invalid, Ether sent is burnt. Compared to bitcoin, there is only one output and not script;

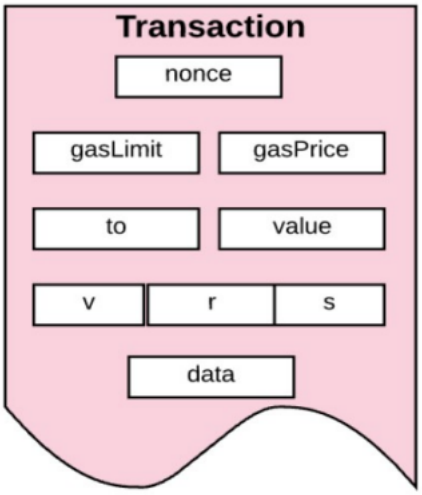


Figure 4.6: Transaction fields

- **value**: the value is directly inserted in the transaction and there is no reference to the previous transaction output. This field is empty for the function invokation but contains a valid value for a payment. The value is transmitted to EOA to be added to the target address account or to contract accounts. In the last case, if no function is found, increase the balance of the contract, otherwise the function named in the data payload must be payable (*can accept Ether from the caller*).
- **Data**: **to contracts accounts**: work as a function selector, allowing to unambiguously identify which function to invoke. The function arguments are encoded according to the rules for the various elementary types. **For contract creation transactions**, it allows to deploy a new contract on the blockchain where data payload contains the compiled bytecode which will create the contract.
- **Gas limit**: maximum amount of gas the sender is willing to pay
- **Gas price**: determined price of a single unit of gas
- **V,R,S**: represent the signature components of an ECDSA digital siginature of the original EOA. Allows to compute the address of the account sending the transaction.

4.1 Solidity

Differently from Ethereum, Bitcoin it's not considered when talking about smart contract because scripts are only to verify the spending of the token, not to support the full execution of user's code.

Solidity is a contract-oriented high level language for the *EVM*. It's a statically-typed, objected oriented language: *smart contracts* are similar to classes. The version of the compiler used it's defined by the `pragma` instruction. The overall lifecycle is:

1. Write code in high-level language
2. Compile to EVM bytecode
3. Deploy witha transaction. The contract code is *read only*
4. Call function with a transaction

Contract example

The purpose of this contract is to create tokens on top of Ethereum, as sort of "layer 2 crypto": only the creator of the contract can mint new tokens and anyone who has an Ethereum account can exchange these tokens.

Differently from coin, **tokens** are build on existing blockchain and are simpler to create because are implemented as smart contracts and leverage security and reputation of the existing blockchain on which are based.

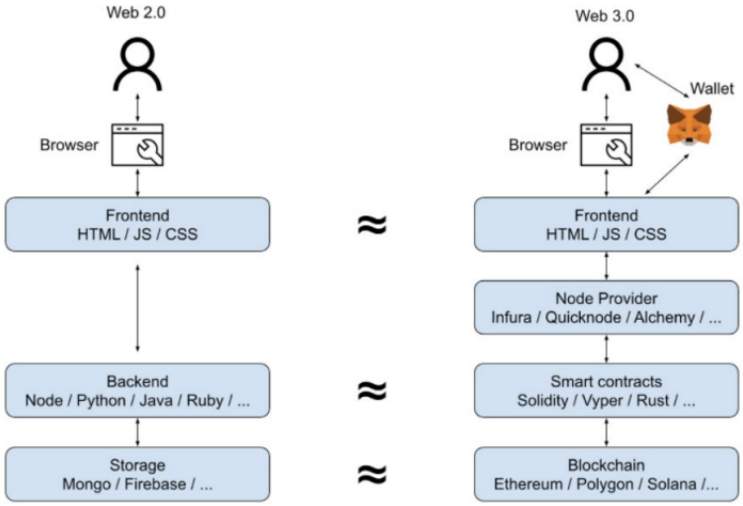


Figure 4.7: undefined undefined

```
        pragma solidity >=0.4.16 <0.9.0;
        contract Token {
                address public minter;

                #associates address to amount of token
                mapping (address => uint) balances;

                constructor() public {
                        minter = msg.sender;
                }
                function mint(address owner, uint amount) public {
                        require (msg.sender = minter);
                        balances[owner] += amount;

                }
        function send(address receiver, uint amount) public {
                if (balances[msg.sender] < amount) return;
                balances[msg.sender] -= amount;
                 balances[receiver] += amount;
         }
        function queryBalance(address addr) public view returns (uint balance) {
                return balances[addr];
        }
}
```

The `constructor()` is autoamtically invoked when the contract is firstly deployed The type of ech variable is specified at compile time and most types can be *cast*, like `bool, unit8, int256, address, string, byte[], mapping(keyType => valueType)` where the lastone is similar to an hashmap.

`pragma` directive The `pragma solidity` directive or *"version pragma"* defines the versions range of the compiler which can be used. This instructs the compiler to check whether its version matches the one required by the pragma: if does not match, the compiler issue an error. So it's used to reject compilation with future compiler version that might introduce **incompatible changes**. The caret `^` before the version number points in `pragma solidity >= 0.4.16 <0.9.0` out that not a compiler earlier version of 0.5.2 and not a compiler after 0.6.0. If no caret is specified all the versions newer than the ofe are referred as ok.

`address` type The address is a 20 byte value represented in hexademical prefixed with $0x$: it does not allows arithmetic operations and it's used to store address of contracts. Remember that, based on

the account type:

- *Address for contract.* defined at the time of contract is created
- *Address of EOA*: hash of the public key of an EOA. The `address` has several properties which can be queried, like the `address.balance` that contains the *wei* of the address, or the `address.code` which contains the code of that address (*that may also be empty*). An address can also be assigned as `address owner = msg.sender` where the right variable is in the *global namespace* thus can be the address that sent the transaction containing the call to the function, either the EOA's address or another contract address.

Data store model

Data in Solidity are of two types:

- **Storage**: variables stored permanently on the blockchain, stored in a Merkle Patricia trie of the block. It uses gas so must be avoided if not necessary.
- **Memory**: stored during function execution, used mainly for temporary variables and defined by the `memory` keyword.

The **storage** model is a sort of *huge array*, initially full of zeroes: each value in the array is 32-bytes wide and there are 2^{256} such values so each state variable declared in teh smart contract will occupy a slot of **depending on its declaration postiion** and its type.

The **memory** model, differently, is a sort of huge array initially full of *zeroes*: variable, structures and array are mapped to the location of this array so the storage is *sparsely populated* and zeros values **are not stored** (*thus an absent key is defined as a mapping to zero*). *Static data* is stored contigously item after item with the first variable, which is stored in slot at position 0, following according to the order or declaration. Thus for each variable, a size in *byte* is determine according to its type: multiple, contigous items that needs more than 32 bytes are packedinto a single storage slot if possible. As an example, the following snippet:

```
contract example {
        uint256 a;
        unit256 [2] b;
        struct exStruct {
                uint256 name;
                uint256 value;
        }
        exStruct c;
}
```

Determine the following slot in the model at **compile time**:

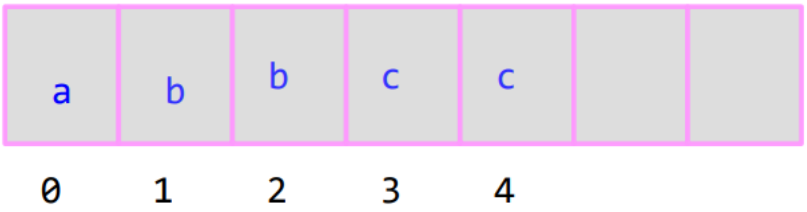


Figure 4.8: Slot determined at compile time

Solidity also have **dynamic types** (*like dynamic arrays*): in this case, for the mapping, you have an empty slot associated with the map: the slot is computed using the `kekka256` hash function, thus the location is chosen at random without experiencing a collision (*there is no allocation and release step from a storage pool*). After computing the hash of the slot, lay the elements of the dynamic arrayas for statically sized-arrays as pictured in 4.9:

The position 5 contains the length while the `kekka256(5)` contains the starting position of the elements.

Consider another code example that involves a *dynamic hashmap* `e`:

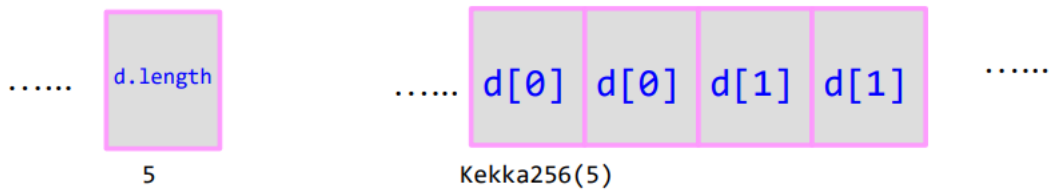


Figure 4.9: Sized array slots

```
contract example {
        uint256 a;
        unit256 [2] b;
        struct exStruct {
                uint256 name;
                uint256 value;
        }
        exStruct c;
        exstruct d[];
        mapping(uint256 => uint256) e;
}
```

To avoid collisions we **concatenate** the key and the mapping slot, hashing the resulting value with `kekka256` (*SHA-3*). concatenard with the position of the empty slot allows to retrieve the associated value.

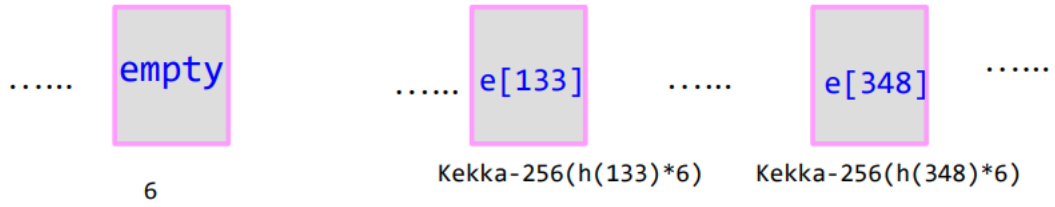


Figure 4.10: Dynamic mapping example slots

The empty slot indicates the starting point of the data structure: the position of the hashed items is obtained concatenating the hash of a given key with the empty slot position. This is like use the entire 256-bit addressable memory space: when the `map` is declared is like that all the possible key exists and every key implicitly bound to all-zero value. The key data is not stored in a mapping, only its hashed value is: the value is stored directly, without double indirection, pointers or linked lists thus the operations are guaranteed to be executed in $O(1)$.

Constructors

Consider:

```
contract Token{
        address public minter;
        mapping (address => uint) balances;
        constructor() public {
                minter = msg.sender;
        }
        ...
```

The `constructor` function is invoked only when initially creating the contract and cannot be invoked afterwards. It's an optional function but if used must be declared with the `constructor` keyword in the new versions. The function is used for customization or setting the initial state of the contract.
The `msg` variable provides information contained in the transaction/message call that actioned the contract: those information are defined in the **global namespace** to provide the context on the transaction that has invoked a function. The function signature is structured as pictured in 4.11.

There are two kinds of modifiers:

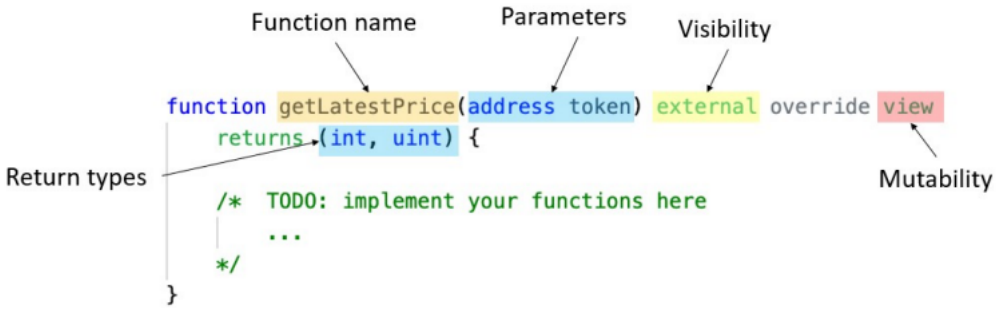


Figure 4.11: Function signature structure

- **Visibility**: define who can **trigger the execution**, defining the behavior of the *caller*, like:
  - `external`: can be triggered by third parties via transacton or from other contracts. (*if `f` is external, `f()` does not work while `this.f()` works*). This modifier is more efficient respect to `public` when receive large array of data.
  - `internal`: can be accessed by the current contracts and inheriting from it
  - `public`: without restrictions (*internal and external*)
  - `private`: accessible only from the contact where ae they are defined adn not by derived contracts
- **Mutability**: define who can access (*and modify*) the data, like:
  - `view` functions: cannot modify state or call other non-view functions
  - `pure` function: cannot read or modify the state, or call other non-pure functions

Error handling Consider:

```
function mint(address owner, uint amount) public {
        require (msg.sender = minter);
        balances[owner] += amount;

}
```

Only the deploter of the contract can mint tokens: `require` tests conditions on function arguments or transaction fields, ensure condtions that cannot be detected until execution time. It throws an error and stop execution if some condition is not true: it's known as *state revert function* because undoes all the changes made in the current call. It consumes the gas up to the point of failure, refunding the remaining gas.

Event logging Events are away for smart contracts written in Solidity to log that something has occurred: interested observers can watch for event and react accordingly: *events* are declared with the `event` jeyword and logged with `emit` keyword. They can also contains parameters.

```
pragma solidity ^0.4.21;
contract Counter {
        uint256 public count = 0;
        event Increment(address who); // declaring event
        function increment() public {
                emit Increment(msg.sender); // logging event
                count += 1;
        }
}
```

Events are an abstraction using the *logging* feature of the EVM: applications in the frontend can subscrube and listen for events as **callbacks**. The event logs are registered in the *transaction log*, a special data structure on the blockchain thus they can also be indexed and filtered. Referring the token contract previously seen, consider to introduce the events in the token contract where the `emit` instruction emit an event each time tokens are transferred:

```
pragma solidity >=0.4.16 <0.9.0
contract Token{
        address public minter;
        mapping(address => uint) balances;
        event Sent(address from, address to, uint amount);
        function send(address receiver, uint amount) public {
                if (balances[msg.sender] < amount) return;
                balances[msg.sender] -= amount;
                balances[receiver]+=amount;
                emit Sent(msg.sender, receiver, amount);


        }
}
```

Example: lottery on blockchain

The main problem is how obtain a *source of entropy* in a deterministic environment. A solution may be:

- *using external information*: this add complexity and external dependencies.
- *using blockchain data*: use the blockhas as a source of randomness but miners can perform an attack and change the timestamp of the block. More complex solution we'll see involves using *RanDAO* or *VRF - Verifiable Random Functions* used in Algorand. The solution here adopted involves using the blockhash to introduce randomization:

```
pragma solidity ^0.8.0;
contract SimpleLottery {
        uint public constant TICKET_PRICE = 1 gwei;
        address [] public players;
        address payable public winner;
        uint public ticketingCloses;
        constructor (uint duration) {
                owner= msg.sender;
                ticketingCloses = block.timestamp + duration;
        }
        function buy () public payable {
                require(msg.value == TICKET_PRICE);
                require(block.timestamp < ticketingCloses);
                tickets.push(msg.sender);
        }
```

The instruction `ticketingCloses = block.timestamp + duration;` takes the **current block timestamp** (*timestamp of block in which the transaction is validated*) returned as UNIX timestamp and add a duration to indicate for how long is possible to buy tickets. The `require` functions are used to ensure that the condition are valid before execute the specific method actions: they usually refer to the *global namespace*. To access the *transaction properties* we can use:

- `msg.sender`: sender of the call/of the message (address)
- `msg.value`: value sent in wei (`uint`) (*the investment in our case*)
- `msg.gas`: gas unused after the execution of the transaction
- `msg.data`: complete calldata (*bytes*) The methods to determine the winner and send the prize are:

```
function drawWinner () public {
        require(block.timestamp > ticketingCloses + 5 minutes);
        require(winner == address(0)); //ensure no winner it still chosen,
        //0x0 as uninitialized address
```

```
        bytes32 bhash=blockhash(block.number-1);
        bytes memory bytesArray = new bytes(32);

        for (uint i; i <32; i++){
                bytesArray[i] =bhash[i];
                bytes32 rand=keccak256(bytesArray);
                winner = payable(tickets[uint(rand) % tickets.length]);
                }
}

function withdraw () public {
                require(msg.sender == winner);
                winner.transfer(address(this).balance);
                }
        fallback () payable external {
                buy();
        }
```

`address(0)` corresponds to the `NULL` value for the address data type. To *flip a coin* in the contract we use `blockhash(block.number-1)`: thie ensure to have the same random value for each node that execute the contract. The function take the hash of the previous block, being determinstic. The computed value `bhash` is used as a form of seed for `keccak256` that compute the 256 hash function. The `winner` variable contains the address of the winner, associated with the `payable` type that indicates that the transfer of the money in the balance of the smart contract must be directed to this address (*either EOA or another smart contract*).

The **global variables** allows also to access to information about the specific block, using the variable `block` or accessing the transaction information using the `tx` variable. In a chain of transaction, this last variable, for example, is able to retrieve the information about the *origin transaction* that actiond the chain. Differently the `msg` global variable allows only to access the information about the last recent transaction that activated the executing function.

Sending and receiving ether There are two main ways for sending Ether:

1. through transaction generated by EOAs
2. transferring Ether between smart contracts This second method can lead to an attack called **DAO attack or reentrancy attack**: because your code may interact with a contract never seend, transferring Ether and executing third party code with an unknown behavior. A solution to this scenario is to **limit the amount of gas** that a function execution can do: this allows to limit the number of action to be executed in a given smart contract. This forbid recursive call. This methods are implemented as:

- `address.trasnfer(value)`: allows gas limit on the execution of the function of the contract receiving funds (*2300 unit*). It throws an error if transfer fails and propagate the failure to the caller.
- `address.send(value)`: return `false` if trasnfer fails but does not throw excetion because the responsibility is delegated to the user to manage failures. The gas limit on the execution of the contract **receiving funds** is setted to `2300` unit and
- `(bool, success, ) = address.callvalue: 10 ether, gas: 10000("")`: this method allows to explicitly set the gas limit so if not specified it's not bounded to any value. It's a lower level function. The 2300 gas limit is arbitrary and can be insufficient for certain trusted operations that must be executed: a limit must be setted always to avoid certain type of recursive attacks.

The operation to **receive Ether** involves using a `fallback` function without an explicit name.

```
fallback () payable external {...} }
```

The `fallback` function is triggered when:

- another contract calls a function in the fallback's enclosing smart contract but the called function name does not mach or exists.

- if marked `payable`, a transaction payment is sent to the contract, without an explicit function call by the sender and no receive function is declared. This function must be *externally called* thus cannot be called from inside the contract in which they are written. When a user sends money to the contract the fallback function is invoked. Respect to the previous lottery example, the `fallback` function is structured as:

```
fallback () payable external {...} }
function buy() public patable {
        require
....
}
```

A *modifier function* allows to verify a given condition, even parametized, to condition the execution of a function and reuse the check in several different function execution.

4.2 Smart Contracts: Security Vulnerabilities

A ***DAO - Decentralized Autonomous Organization*** is an organization in whose management is decentralized and decision are autoamtized though smart contracts thus rules are encoded in a smart contract, removing the need for a central governing authority. The main goals of this type of organization is to reduce costs and provide more control and access to investors.
A *DAO* is a platform that allows everyone with a project to pitch their idea to the community for funding. Investors from around the world can send `ETHER` to a unique wallet in exchange for **DAO Tokens**: thosetokens give voting rights so the funds are pooled together.
If a proposal passes a preliminary's curator checks, owner of a DA tokens can vote it, proportionally to their token: if a proposal is approved by a *quorum* of all tokens, the DAO automatically trasnfer *Ether* to the smart contract that represents the proposal. Finally, if the proposal increment the project value, stakeholders receive their rewards.
Basically a DAO is a *complex smart contract* with many features to reflect laws, respecting the stakeholder's rights, guaranteeing the right of a stakeholde rto require his shares to have back his funds when a prposal they do not want be a part of get approved despite their objection (*Appraisal right*).

Simplified DAO

Imagine the following simplified scenario:

Figure 4.12: Simplified DAO scenario

It's noticeable to highlight that the *wallet* contract does not use the function `send` or `trasnfer` to send *Ether*: it uses a previous Solidity version of the call function, named `call.value`, which does not specify any gas restriction. The `withdraw` method, the first time it's called, allows to transfer 10 *Ether* from the wallet: if a second call is made, no transfer can happen because the `balance` is updated to 0. This simple scenario allows to describe a set of vulnerabilities that lead to an attack in which the user contract can withdraw more than 10 *Ether*.

4.2.1 1. Re-entrancy vulnerability

A *re-entrancy* is a scenario in which an attacker performs **recursive withdrawals** to steal all *Ethers* locked in a contract. A procedure is *re-entrant* if its execution:

- can be *interrupted* in the middle
- *iniated over (re-entered)*
- both runs complete without errors For a contract, a *non-recursive*, **re-entered** function allows the attacker to exploit the ***fallback/receive*** function to re-enter the function. The *atomicity* and sequentiality of transaction may induce programmers to believe that it cannot be re-entered. In general, *re-entrancy* may result in unexpected behavior and *loops of invocations* which eventually consume all the gas.

The below code presents the `InsecureEtherVault` contract, providing a simple vault allowing users to deposit their Ethers, transfer deposited Ethers to other users, withdraw all deposited Ethers, and check their balances.

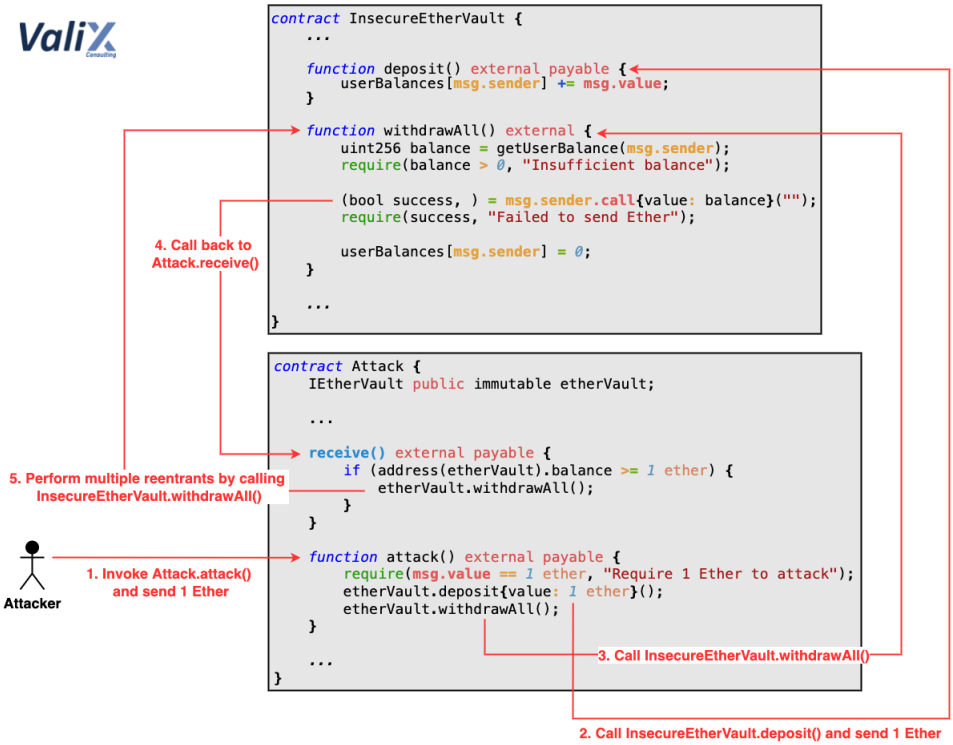


Figure 4.13: InsecureEtherVault Solidity code

As soon as the low-level function `call` is executed, a number of Ethers indicated by the `balance` variable would be sent to the user wallet or external contract (Step 4). If an attacker's `Attack` contract is the recipient, the contract can do the reentrancy by recursively calling the `withdrawAll` function (Step 5) to drain out all Ethers locked in the `InsecureEtherVault` contract.

The attack is effective here because the `call` function is executed before updating the withdrawer's balance to 0 (i.e., `userBalances[msg.sender] = 0`). Consequently, the `Attack` contract can interrupt the control flow in the middle and execute the loop calls to the `withdrawAll` function. **Since the `withdrawAll` function would still retain the balance before the update, the `Attack` contract can steal all Ethers.** (*See Solidity Security By Example: Reentrancy for the full description and patching*)

The solutions to the proposed attack method are:

- to use the functions `send()` or `transfer()` instead of `call.value()` (*in `InsecureEtherVault`*) or to **manually limit** the amount of gas passed to `call.value()`. This last method would not allow for recursive withdrawal calls due to the low gas stipend.
- to *update the user balancer prior to the transfer:* any recursive withdrawal call would attempt to trasnfer a balance of 0 Ether. A general rule is that if no *internal state updates* happen after an

Ether trasnfer or an external function call inside a method then the method is **safe from the re-entrancy** vulnerability.

4.2.2 2. Arithmetic over/under flow

Consider the following contract:

```
pragma solidity ^0.8.0;
contract Token {
                mapping(address => uint) balances;
                uint public totalSupply;
        constructor (uint initialSupply) {
                        balances[msg.sender] = totalSupply = initialSupply;
        }
        function transfer(address to, uint value) public returns (bool) {
                        require(balances[msg.sender] - value >= 0);
                        balances[msg.sender] -= balances[msg.sender] - value;
                        balances[to] = balances[to] + _value;
                        return true;
        }
        function balanceOf (address owner) public view returns (uint balance) {
                        return balances[_owner];
        }
}
```

In the previous code the `require(balances[msg.sender] - value >= 0);` check allows to verify that the `value` can be subtracted from the sender's `balance`, but still allows to perform the attack. The *vulnerability* is created by an operation requiring a **fixed-size variable** to store a number that is outside th range of the variable's data type like:

```
uint8 x = 0;
uint8 y = x - 1; //y is 255
```

In this case, the `uint` indicates an *unsigned integer of 8 bits, non negative number* thus subtracting 1 from `x` results in an underflow that result in 255. This scenario can be exploited in the `trasfer` function because if the attacker has a 0 balance, the function is called with `_value` as a non-vero value that allows to obtain a positive value and thus pass the `require` check so thebalance will be updated with a positive number. To avoid this scenario, it's better to not use the Solidity arithmetic operators directly but use third-party library like **SafeMath library**.

4.2.3 3. Front Running attack

Ethereum nodes pool transactions and form them into blocks: all the transactions are visible in the *Mempool* of a full node for a short period before they are executed (*thus inserted in a block by a miner*) so they can be *observed* by nodes to potentially perform an attack. The **miner** who solves the block also chooses which transaction from the pool is elegible to enter the block: typicaly the order is given by the `gasPrice` of each transaction. The **attacker** can observe a transaction T before it's commited ina block so he can create a transaction T_i of their own that pays a *higher gas price*, allowing T_i to be executed before T: the order of the transaction can influence the final price payed when the transaction is executed thus it's possible that T_i may indirectly alter the cost payed by the transaction T.

Possible solutions to prevent this scenario are:

- *upper bound* on `gasPrice`: prevents users from getting preferential transaction ordering but does work if the **miners are also the attacker**.
- use a *commit-reveal scheme*: transaction is sent with hidden information so after the transaction is included in the block, the user send another transaction revealing the data that was sent.

4.2.4 4. Transaction ordering attack

This type of attack is a **race condition attack**: if you purchase an item at a price advertised, you expect to pay that price. This attack will *change the price during the processing of your transaction* because some one else (*the contract owner, miner or another user*) has **sent a transaction modifying the**

price before your transaction is complete. Two transactions can be sent to the *mempool/tx-pool*, the order in which they arrive is irrelevant: the attacker could be a **miner**, as the miner can choose the order in which the transactions are mined, or may be a **node inserting a transaction with a higher fee**. A problem in Smart Contracts that rely on the state of storage variables to remain at certain value according to the order of transactions. Here an example:

```
pragma solidity ^XXXXXX;
contract TransactionOrdering {
        uint256 price;
        address owner;
        event Purchase(address _buyer, uint256 _price);
        event PriceChange(address _owner, uint256 _price);

        modifier ownerOnly() {
                require(msg.sender == owner);
                _;
        }
        constructor TransactionOrdering() { // constructor
                owner = msg.sender;
                price = 100;
        }
        function buy() returns (uint256) {
                Purchase(msg.sender, price);
                return price;
        }

        function setPrice(uint256 _price) ownerOnly {
                price = _price;
                PriceChange(owner, price);
        }
}
```

The *owner* of the contract deploy it with `price = 100`: the buyer of digital asset call the `buy()` function to set a purchase at the price specified. The contract owner calls `setPrice()` and update the `price` storage value to 150. The contract owner sends the transaction with aa *higher gas fees* so it's mined first, updarting the state of the contract due to the higher gas fee. The buyers transaction gets mined soof after but not the `buy()` function will use the new update price of value 150.

```
pragma solidity ^0.8.0;
contract Roulette {
        uint public pastBlockTime; // forces one bet per block
        constructor() payable {} // initially fund contract

        // fallback function used to make a bet
        fallback () external payable {
                // must send 10 ether to play
                require(msg.value == 10 ether);

                require(block.timestamp != pastBlockTime);
                // only 1 transaction per block
                pastBlockTime = block.timestamp;
                if (block.timestamp % 15 == 0) {
                        // winner address
                        payable ap = payable(msg.sender);
                        ap.transfer(address(this).balance);
                }
        }
}
```

It's like a *simple lottery*: one transaction per block can bet 10 ether for a chance to win all the balance of the contract. The *basic assumptions* are:

- `block.timestamp` last two digits are *uniformely distributed*
- there would be a 1 in 15 chance of winning this lottery The **miners attack** consists in the miners adjust the timestamp: they can choose a timestamp such that `block.timestamp` *modulo* $15 = 0$. In this way they may win both the *Ether locked in this contract* and the *block reward.*

In practice, miners **cannot choose arbitrary timestamp**, they must be *monotonically increasing* thus block times cannot be set too far in the future. In any other case, the block will be rejected by the network.

4.2.5 5. Phishing attack

Generally speaking, a phishing attack consists in the fraudulent attempt to obtain sensitive information like username, password, credit card details by appearing as a *trustworthy entity* in electronic communication. A phishing attack in Solidity may be perfomed by exploiting the global variable `tx.origin` (*address of account that generated the transaction, the original external account*) instead of `msg.sender` (***immediate account**, external or contract that invokes the function*) :

- The attacker calls the target contract directlu thus his authorization is checked based on his *personal address*
- The attacker creates hiw own contract calling the target contract, runs a phising campaign to run the functionality on the attacker's contract.
- The attackers contract makes a transaction call to the target contract but the call originates from the user's address via `tx.origin`
- If the target contract processes the transaction via `msg.sender` the authorization check will be denied, forbidding to authorize the operation from the attacker
- If the target contract checks authorization via `tx.origin`, the attacker is accessing the target as the victims address thus ca bypass any authorization checks and simply process functionality as the victim user. Here an example:

```
pragma solidity ^0.8.0;
contract Phishable {
                address payable public owner;
                constructor (address payable _owner) {
                        owner = _owner;
                }
                fallback() external payable {} // collect ether
                function withdrawAll(address payable recipient) public {
                        require (tx.origin == owner);
                        recipient.transfer(address(this).balance);
                }
}

import "Phishable.sol";
contract AttackContract {
        Phishable phishableContract;
        address payable attacker; // The attacker's address to receive funds
        constructor(Phishable phishableContract, address payable attackerAddress) {
                phishableContract = phishableContract;
                attacker = attackerAddress;
        }
        fallback () external payable {
                phishableContract.withdrawAll(attacker);
        }
}
```

The **attacker**:

- deploys the `AttackContract`

- convince the owner of the `Phishable` contract to send the *Attacker Contract* some amount of Ether
- the `fallback` function is invoked
- in turn, the `withdrawnAll` is invoked

The **victim**:

- receives a call to `withdrawnAll`
- the address that first initialised the call was the victim, that is the owner of the `Phishable` contract.
- therefore, `tx.origin` will be equal to owner and the require of the `Phishable` contract will pass.
- the victim sends all the funds to the attacker

4.3 Tokenization

Tokens are *coin-like* objets generalli lacking a legal framework: they are issued by a private entity for a specific use. The value of those tokens may be high but only within the community that makes use of them. There are two main types of tokens:

- **Fungible**
- **Non-fungible (NFT)**

The *fungibility* of a token can be defined as *"being freely exchangeable or replaceable, in whole or in part, for another of like nature or kind"*. This implies main properties like:

- **Interchangeability**: each token is interchangeable with other tokens of its same kind but there aren't defined features of that specific token.
- **Merging**: can merge unit of fungible asse to get a higher value in quantity
- **Divisibility**: can sen or receive a fraction of a token

Differently, a **non-fungible** item is when two items may look identical at a glance but each will have *unique information or attributes* that makes them irreplaceable or impossible to swap. They are usually charaterized by *scarcity* and *originality*. In the category of *NFT* there is a sub-category for **collectibles** that can include *real-world collectible* (*like Pokemon trading card*).
In the area of **Fungible tokens** we can identify two differen types:

- **Utility token**: application token or user tokens offered by a company giving future access to products or services. They are not an investment but better as a coupon.
- **Security token**: as an hybrid between shares of a company and cryptocurrency. Gives the owner tights and obligation with voting rights and dividends, thus companies sell shares in form of cryptographic token.

The advantages of *security tokens* are that the voting rights are realted to the amount of tokens owned: they can be offered throigh an *ICO - Initial Coin Offer*. The advantages of *utility tokens* are evident when a company want to obtain financing for a project: it can create a *coupon* that can be redeemed in the future for access to its services.

ICO - Initial Coin Offer

An ICO is basically a cryptocurrency version of a *crowdfunding campaign*. The developer issue a *limited amount* of tokens, ensuring that the token itself have a value and the ICO has a goal to aim for. The price can be statically *pre-determined* or may increase or decrease depending on how the crowd sale is going. The buying process follows those steps:

- if someone wants to buy the tokens they send a particular amount of Ether to the crowd-sale address.

- when the contract acknowledges that this transaction is done, they receive their corresponding amount of tokens.
- ICO is successful if it is really well-distributed and a majority of its tokens is not owned by one entity

Despite the term **fungible token** and **currency** are used interchangeably, the technology is different because the first are usually implemented on the blockchain of the second one.

Semi-fungible token (SFT) Alonside the *fungible* and *non-fungible (NFT)* tokens, a third type is known as **Semi-fungible Token (SFT)**: it can *change* the state from being *fungible* to *non-fungible*. An example is represented by a *ticket* for the final football championship match:

- interchangeable for any other ticket of the same match, and same seating class
- after the match is over, can no longer be used for entering to the stadium
- may become a collector's item for fans with a different value assigned to it.

4.3.1 ERC Tokens

ERC (Ethereum Request for Comments standard define smart contract that have:

- a **pre-established standard structure** for storing and managing tokens
- a set of **pre-defined functions** that can be executed on the token
- give developers the guarantee that assets will behave in a specific way and describe exactly how to interact with the basic functionality of the assets.

The most widespread standard is **ERC-20**, now used as standard for *Fungible tokens (FTs)*, while the **ERC-721** is used for *NFTs* and **ERC-1155** is for semi-fungible tokens. Those three standard define different way of mapping addresses to data it represent:

- ERC 20 maps `addresses` to `amount`
- ERC 721 maps unique `IDs` to `owners`
- ERC1155 has a nested mapping of `ID` to `owners` to `amount`

ERC-20 tokens

The **ERC-20 tokens** allows to implement a **sub-accounting system parallel to the Ethereum** main ledger, having their own unit of account. There is no mixing with the Ether balances of the addresses while, at the same time, guarantees the transparency, traceability and security provided by the main Ethereum network.

A ***token contract*** contains a map of account addresses and their **balance**: the concept of balance may vary depends from the token contract (*it may represent an amount of object, right, monetary values, etc*).

The ***mandatory functions*** the standard define are

- `totalSupply`: returns the total units of tokens that currently exist in the token smart contract
- `balanceOf`: given an address (*user*) returns the token balance of that address
- `transfer()`: given ana ddress and amount, trasnfer the tokens to that address from the balance of the address that executes the transfer. It's a **single-step transaction**: just like a conventional crypto transaction between two wallets.
- `transferFrom()`: allows to transfer tokens from an address that is not that you own
- `approve`: it's used in tandem with `transferFrom` function, allowing the owner to approves a delegate address to withdraw tokens to trasnfer to other account. It define also the **maximum amount of allowed token** to trasnfer.

- `allowance`: implement a **two-step payment process**, allowing a third party to carry out a transaction of token on your behalf. It's used for cyclic payments (*like subscription*). It checks also the current *approved number of tokens by an owner to that specific delegate* (*the max setted by* `approve` *function*). while the ***optional fields*** defined are:
- `name`: return human readable name of the token
- `symbol`: human-readable symbol for the token
- `decimal`: number of digits (*up to 18*) that come after the decimal place, when displying values on the screen. If the decimal is 2, the token amount is divided by 100 to get its visual representation. This field is required because Ethereum does not work with decimal number but only integers.

Allowance example Consider user `A` with 100 tokens and user `B`. `A` want to give permission to `B` to spend all 100 tokens, so:

1. `A` calls `approve(address(B), 100)`
2. `B` checks how many tokens `A` gave him permission to use by calling `allowance(address(A), address(B)`
3. `B` sends to his account some of these tokens by calling `trasnferFrom(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.

ERC-721 Interface

As mentioned, the **ERC-721** is the Ethereum token standard for *NFTs* on the Ethereum blockchain. The *main functions* to be implemented are:

- `ownerOf(ID)`: each ERC721 token is referenced on the blockchain via a unique `ID`. It return the owner's address.
- `transferFrom`: transfer ownership of an NFT. The caller is responsible to confirm that the receiver is capable of receiving the NFTs, otherwise they are **permanently lost**.
- `approve`: approves, or grants, another entity permission to transfer one of the tokens on the owner's behalf
- `SetApprovealForAll`: enable or disable approval for a third party to manage all the assets of the user (address) while executes this command
- `SafeTransfer`: transfer the ownership of a NFT from one address to another address
- `TransferFrom`: transfers ownership of an NFT, under the hypothesis the receiver is capable of receive NFTs.

Other information defined in a NFT contract are `name`, used to indicate the name of the token to external contracts and `symbol` that provides a token's shorthand name. Other token-specific information can be associated to an NFT by using the ***metadata*** section that provide nformation for a specific token `ID`. The metadata section can contains *image, description, preview content, traits, unlockable content, supply, royalty etc.*

The **ongoing royalties** metadata, if setted, when the NFT is sold in the future, allows a certain percentage to go back to the original creator automatically. The creator is able to indicate the address to which the royalties will be sent, The **unlockable content** metadata contains content that only the owner of the NFT can see or access.

Storing metadata It's possible to store the metadata in the smart contract, thus **on-chain data** or host it separately, obtaining **off-chain data**. The **on-line** storage may be extremely expensive and not recommended, except for:

- *information that must be persistenly recorded on-chain* (*digital art*)
- on-chain logic must interact with metadata (*e.g. age of Cryptokitties influence how kitty can breed*)

Differently, the **off-line** storage can be implemented in two different way: using a *Cloud provider* or using *IPFS - InterPlanetary File System* that is a decentralized peer-to-peer network of computers around the world where data content is stored across multiple locations.