(Tools for) IPFS

A **hash pointer** is a pointer to where some info is stored and a cryptographic hash of the info. If we have a hash pointer, we can ask to get the info back or verify that it hasn't changed.

So the idea is to build data structures with hash pointers, like the blockchain that it's a linked list with them. This allows *tamper-evident logs*, where if someone tampers the *k-th* block of the chain, the *k-th+1* block won't match up: a structure is tamper-free if the head of the list is stored in a place that an adversary can't change *(trusted location)*. ***Note:*** *to hash a block, you have also the hash pointer itself!*

Evolving from a linked list, we find binary trees and they can be used with hash pointers too: **Merkle trees.** For the previous reason, the root needs to be in a secure place. So we can use hash pointers in any pointer-based data structure if that structure has no cycles.

Bloom Filter *(probabilistic data structure)*

Consider the set S = {s1, s2, …, sn} chosen from a very large universe U, we need a data structure supporting queries like "is k an element of S?" that both are computed efficiently and uses reduced space (-> *false positives)*

S may be many things, like a set of keyboards, a set of pieces of a file owned by a peer or even a set of bitcoin addresses: it'll be lightweight and sending the Bloom Filter to nodes will save a lot of bandwidth.

Note: *lower the space, higher the errors*

m bits set to 0, k universal hash functions, calculate f(x), g(x), h(x), … and put 1 where the result is. To find an element, apply all the k functions again and see if there are only 1s (-> maybe yes), one 0 -> surely not.

The probability of false positives is $(1 - e^{-kn/m})^k$

A Bloom Filter becomes effective when m = c*n, with c constant *(low value, like c = 8)*

bits/element	m/n	2	8	16	24
number of hash functions	k	1.39	5.55	11.1	16.6
false-positive probability	f	0.393	0.0216	$4.59 \cdot 10^{-4}$	$9.84 \cdot 10^{-6}$

So there's a trade-off between space and number of hash functions.
If m and n are fixed, k = m/n ln2 is the best value to minimize the probability of k, leading to a 0.62^(m/n)
Reminder: *an optimal Bloom Filter has half of the bits equal to 1 and half equal to 0*

Operations

- **Union:** *obtained by computing the bitwise OR bit of BF1 and BF2*
- **Delete:** *must be done on a Counting BF (inserting -> counter++, deleting -> counter--)*
- **Intersection:** *needs the same number of bits and the same number of hash functions, it's the bitwise AND of BF1 and BF2 (it approximates the intersection of the two sets)*

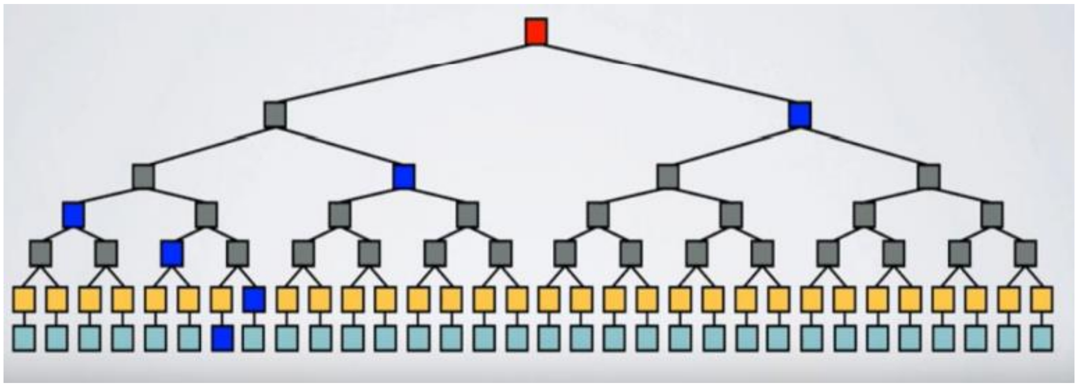
http://www.docjar.com/html/api/org/apache/cassandra/utils/BloomFilter.java.html

Merkle Tree

A data structure summarizing information about a big quantity of data with the goal to verify the correctness of the content. It's simple, efficient and versatile, exploiting a hash function H *(<u>on leaves</u> H is applied to the initial block, <u>on internal nodes</u> H is applied to the concatenation of the hashes of its sons)*

<u>Example:</u> *A wants to prove that the data sent to B is not tampered, so it sends a block <u>and</u> the hash of all other blocks. B gets from a trusted source the root hash of all the data, hashes the sent block, hashes the string resulting by appending all the hashes and if it's equal to the hash gained by the trusted source -> OK.*

Note for efficiency: take one block per level only (so 4 blocks for 16, or 6 for 32, etc) *(log(n) hashes)*



Note: it might be a perfect Merkle tree *(all leaves are on the same lvls, each node has 0 or 2 children)*. This implies a number of properties like: *#nodes = 2^n* *#leaves = (N+1)/2* *depth = log(L) + 1*

Cons of this structure is that it's static: the smallest segment size is the smallest unit of verification. (?)

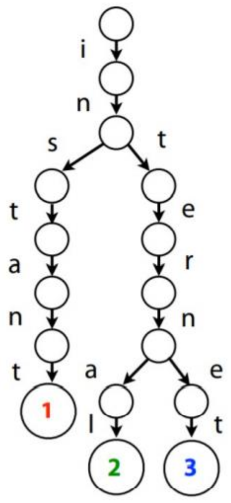
This data structure is used in P2P applications to help ensure that data blocks received are undamaged/unaltered: the site indexing the *file.torrent* stores the root hash and the total size of the file/pieces, being the trusted third party.

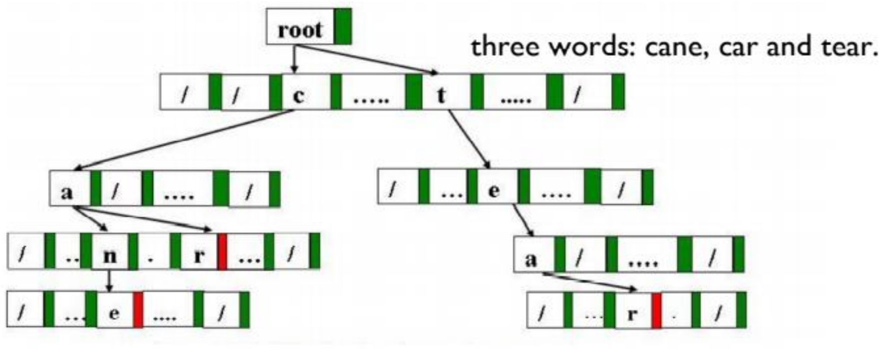
Trie

Coming from "*retrieval*", it's a tree representing a collection of strings with one node per common prefix. Very useful when keys are strings

Each node corresponds to some string given by the path traced from the root.

Each node stores a flag indicating if the string is in the set and an array of pointers





Cost to lookup a string w is to follow at most |w| pointers *O(|w|)* **Note:** <u>independent on #strings!</u>

Depending on the way we store characters, we get different complexities:

1. Array of child pointers of size of the alphabet Σ: LU O(1) but wasting space
2. Hash Table LU O(1) and still wasting space, but less
3. List of child pointers compact, but LU O(Σ) in the worst case
4. Binary Search tree compact, LU O(log Σ) in the worst case

In any case, the **insertion** is a normal lookup, adding new nodes as needed and setting the flag in the final node. **Removal** depends if the node has no children (remove that node) or not.

Although time-efficient, Tries can be extremely space-inefficient O(N * | Σ |)

➔ **Patricia Tries** *(Practical Algorithm To Retrieve Information Coded In Alphanumeric)*

Eliminate unnecessary trie nodes, because having chains of one-child nodes is a waste *(compressed tries)*.

Interplanetary File System

(Juan Benet 2013, IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3))

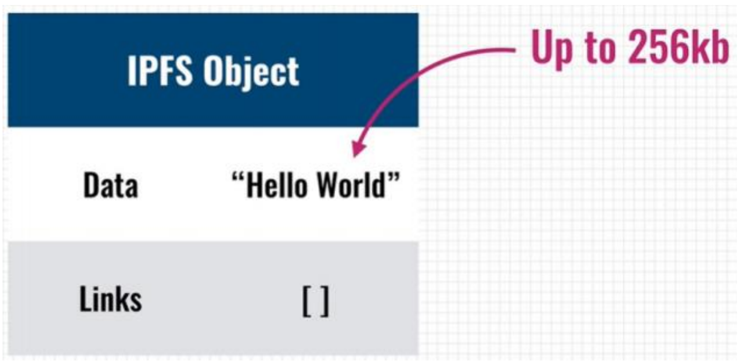
In the current Internet, we have the information collected by big servers *(a.k.a. centralized systems).* This has many downsides, like censorship, poor bandwidth optimization, etc.

➔ **IPFS Project:** *"make the web faster, safer and more open"* *https://github.com/ipfs/go-ipfs*

We already spoke about the differences between Location-based addressing & Content-based addressing. IPFS combines DHT, incentivized block exchange and self-certifying namespace.

It leverages and combines many successful P2P system ideas:

- **Routing:** DHT + improvements for security *(S/Kademlia)* and performance *(sloppy hieratical DHT)*
- **Block Exchangers:** BitTorrent
- **Version Control Systems:** Git
- **Self-Certified File Systems:** SFS



Links are hash pointers *(content-based addr)* to other IPFS objects. This type of addressing means that when a content is added it can't be changed!

But what about mutable data? IPFS supports **versioning** of files *(keep all the story of the web content)*

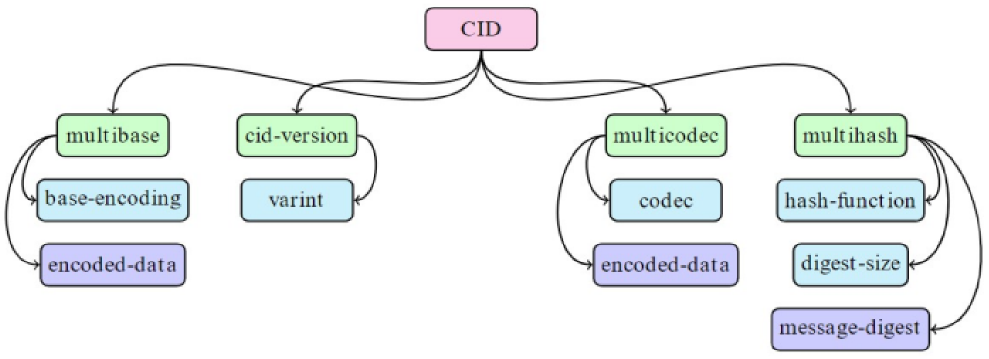
Note: *there's data deduplication to eliminate duplicate copies of repeating content (hash allows this).*

How to manage the naming system for IFPS? Since hash links to contents are not human-readable *(and immutable)* we use an **Interplanetary Naming System (IPNS)**, a system allowing to always point to the latest version of content that is human-readable and easy to remember *(like a DNS)*.

To know where the files are stored in IPFS, each node of the network keeps a cache of the files it has downloaded and shared: it's a BitTorrent-like swarm, **but** a single swarm for all IPFS, not a swarm for each content. As usual, if all the owners go offline there's unavailability, so we can guarantee several copies in the network, etc.

➔ **FileCoin:** built over IPFS, a decentralized market for storage *(free disk space? Make money for it!)*

To address the contents, we introduce the **IPFS Content Identifier (CID)**. It contains the hash value of the referred object + multiple metadata fields. It's obtained as a concatenation of multiformat objects *(objects containing self-describing data)* in order to allow easy future extensions of protocols/data.



Current protocols:

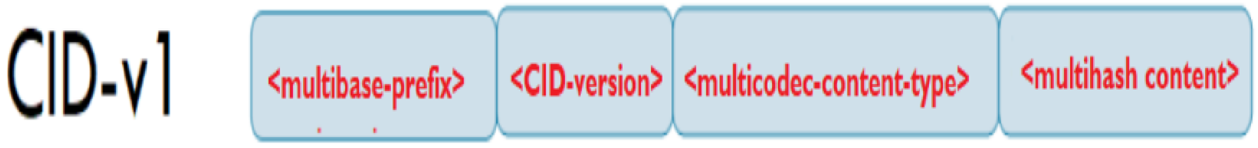
- **Multihash:** *self-describing cryptographic hashes*
- **Multibase:** *self-describing base encoding*
- **Multicodec:** *self-describing serialization* *github.com/multiformats/multicodec*
  Hash functions** (sha2-256, ..), **serialization functions** (JSON, ..), **Internet protocols
- **Multiaddr:** *self-describing network addresses*
- …

This is important because we can't tell if we will need different hash functions in the future (today ones might be broken like MD5) and the same happens for network protocols (HTTP/1 -> HTTP/2). Furthermore, this guarantees that there will be interoperability, no "lock-in" and the data format values will be as small as possible.

Version 0: *https://cid.ipfs.io/*

all CIDs v0 are 46 chars long starting with *"Qm"*

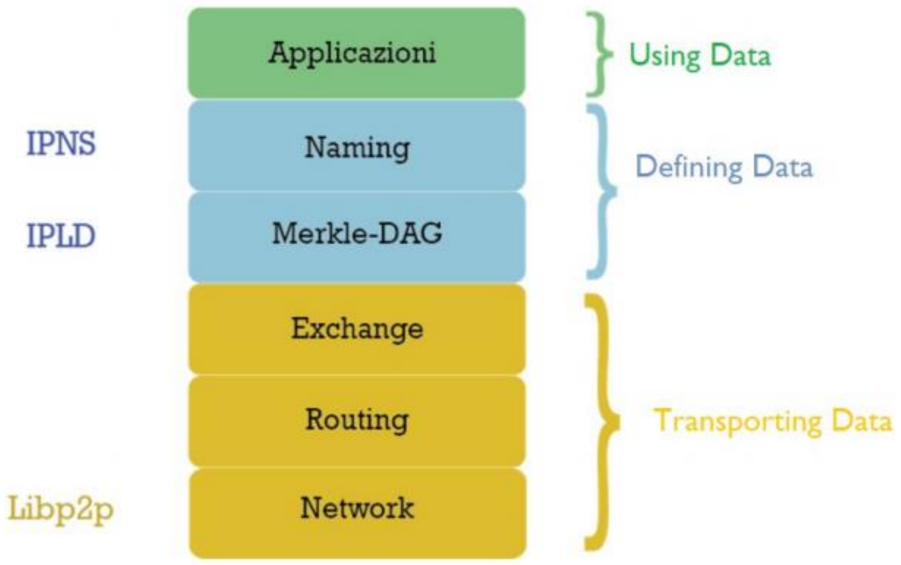
Version 1:



Multibase: a prefix for the base: *b = base32* *z = base58* *f = base16*

Multicodec: uniquely identifies the encoding method used

The IPFS stack



LibP2P

The LibP2P library allows the generation of the *PeerId.* Each peer controls a pair *<public key, private key>*, and the *PeerId* is the cryptographic hash of a peer's *public key.* The PeerId may be embedded in a CID since it's represented by a multihash but it may be also encoded in structures called..

➔ **Multiaddr**
A convention for encoding multiple layers of addressing information into a single "future-proof" path structure. It's human-readable and machine-optimized, allowing many layers of addressing to be combined and used together. **It allows recursive encoding** *(nested protocols).*
Example: */ip4/127.0.0.1/udp/1234 encodes both the IP level and the Transport level*

Routing

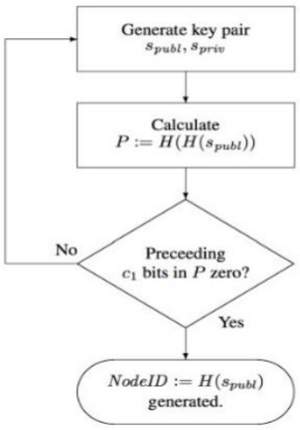
Use the hash of the file as a key to return the location of the file. Once the location is determined, the transfer is a decentralized one (P2P).

Case Study (S/Kademlia): a secure key-based routing protocol based on Kademlia, with high resilience against common attacks. The security improvements depends on:

1. **the limitation of free node identifier generation** *(PoW through cryptopuzzles)* and public key cryptography. *The required computational effort deters the DoS attacks, because users can access to a particular service if they solve a highly computationally expensive problem (the check must be easy though!). Given a cryptohash function H and* $\oplus$, *the* ***cryptopuzzle*** *may be:*

a. ***Static*** *if generating key K so that c first bits of double hash H(H(key)) = 0*

choose NodeId = H(K) ***-> NodeId can't be chosen freely***



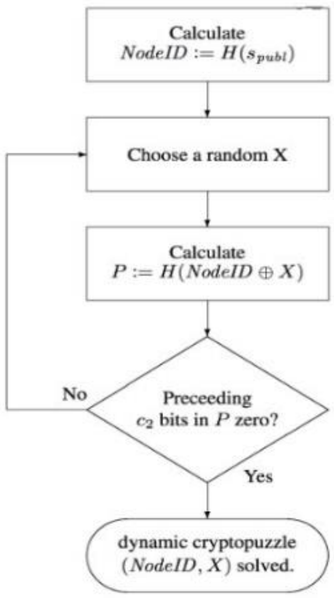
Generate key pair s_{publ}, s_{priv}
Calculate $P := H(H(s_{publ}))$
Preceeding c_1 bits in P zero?
No
Yes
$NodeID := H(s_{publ})$ generated.

Prevents eclipse attacks*, because it makes it difficult to generate a specific non-random node identifier.*

"Find the public key PK such that the double hash of PK has c1 leading zero-bits", where c1 is considered constant and the size of the public key must be increased subsequently.

b. ***Dynamic*** *if generating X so that c2 first bits of H(key $\oplus X) = 0$*

Increase c2 over time to keep NodeId ***generation expensive***

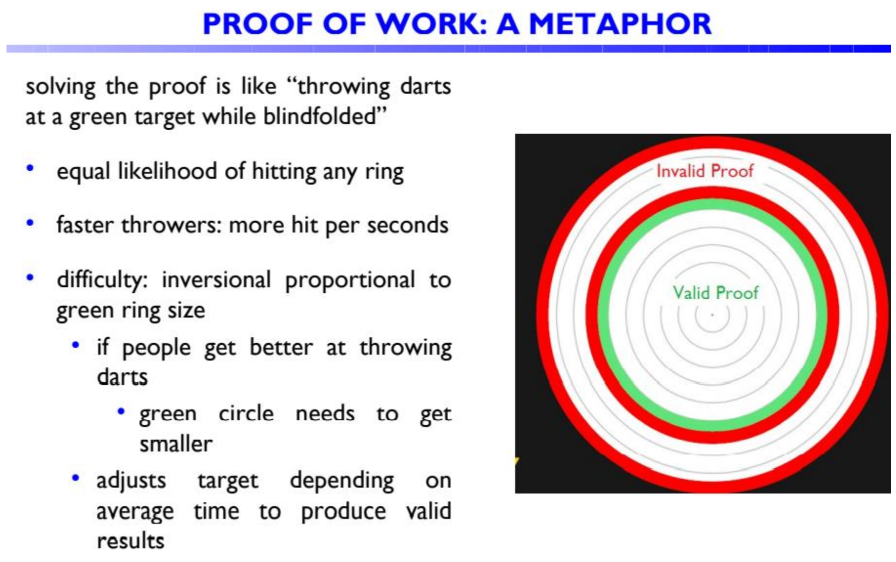


Calculate $NodeID := H(s_{publ})$
Choose a random X
Calculate $P := H(NodeID \oplus X)$
Preceeding c_2 bits in P zero?
No
Yes
dynamic cryptopuzzle $(NodeID, X)$ solved.

Prevents sybil attacks*, due to a lot of computational effort to generate many identities.*

"NodeID is in the hash of the public key PK, to avoid the generation of large amounts of nodes, choose a random X, then the hash of $\oplus$ of the PK and X must have c2 leading zero-bits"

Verification cost: O(1) ***Creation cost: O(2^c1 + 2^c2)***



PROOF OF WORK: A METAPHOR

solving the proof is like "throwing darts at a green target while blindfolded"

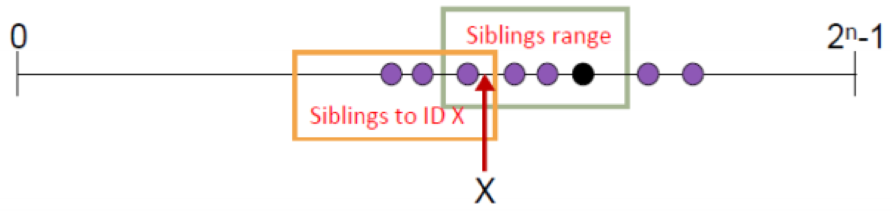
- equal likelihood of hitting any ring
- faster throwers: more hit per seconds
- difficulty: inversional proportional to green ring size
  - if people get better at throwing darts
    - green circle needs to get smaller
  - adjusts target depending on average time to produce valid results

2. **Parallel look-ups** over multiple disjoint paths

   In order to avoid an attacker rerouting packets into its own subset. Iterative look-ups allow to ensure look-ups on disjoint paths: first look-up k closest nodes in the own routing table, distribute them over d path look-ups and do parallel path look-ups (<u>check if a node was already visited)</u>

3. Kademlia's **routing table's extension** by a sibling list

   *Siblings are nodes where the Kademlia information is replicated. The reliability of sibling information arises when a majority decision on information stored in the DHT is needed to compensate for adversarial nodes attacks. Each node needs to know s closest nodes [the siblings] to an identifier if it falls inside its sibling range. The sibling list size must be at least >5*s to ensure that the node knows at least s siblings to an identifier in its siblings range w.h.p. [see paper for formal proof]*



[Baumgart, Ingmar, and Sebastian Mies. "S/kademlia: A practicable approach towards secure key-based routing." IEEE Int. Conference on Parallel and Distributed Systems. 2007]

Content Distribution Networks (CDN)

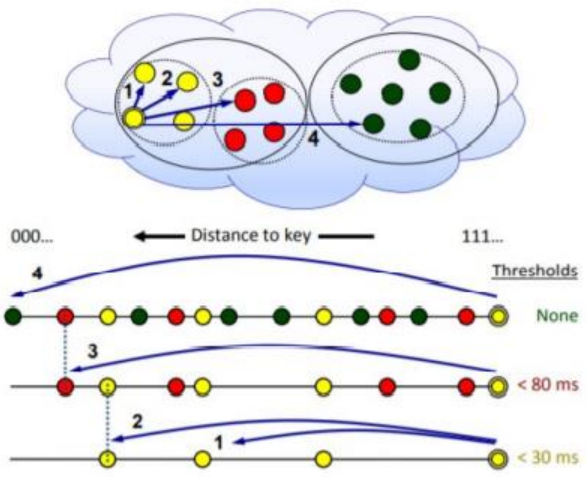
Geographically distributed network of Web servers around the globe, to improve the performance and scalability of content retrieval. It allows several content providers to replicate their content in a network of servers.

Example (Coral)

Goal implement a client caching scheme

How to develop a P2P CDN by using Sloppy DHT [*solves hot spots and poor data locality]* to avoid hot spots on caching nodes and finding nearby data without querying distant nodes. Using the URL's as a key, the value corresponding will be the list of nodes that cache that web page. But an URL may be more popular than others *(hot spot)* and it will quickly overload and a query may travel all the world before learning that the content is cached close to the requester. But a node doesn't need to know every location of a resource, it only needs a valid nearby copy! (that's the Sloppy DHT idea) so each node stores only a maximum number of values for a particular key, so the values associated with a key are spread on multiple nodes.

To **insert content** you store the data locally and insert a pair *<content key, reference to node>* in the DHT and approach to the responsible node through Kademlia routing: if the closest node is full, **backtrack one hop** on the look-up path and store on the first free node. *Keys will be stored on "close enough" nodes too!*



This way we can group nodes into clusters *(same nodes have the same ID in all the clusters).*

If no key is found at lvl X, go to level X-1 and continue, then you'll eventually reach lvl 0 which covers the full network

[Sloppy hashing and self-organizing clusters: Michael J. Freedman and David Mazieres]

Exchange Level

(Slides on NAT)

Bitswap

A protocol for data (blocks) exchange. *Files are split into blocks, the smallest unit of transferable data.*

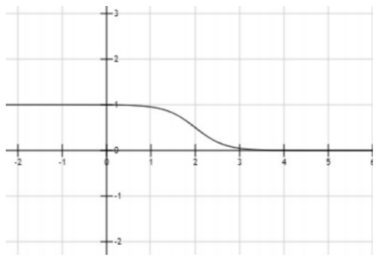
Remember that in IPFS there's a *single swarm of peers* for all the data shared by the user, opposed to the BitTorrent which has a separate swarm of peers for each file.

Note: You can monitor the single swarm by the IPFS CLI interface through the command: *ipfs swarm peers* returning the multiaddresses of the peers.

The basic strategy is based on keeping track of the amount of data two peers share and receive, *respectively building credit and building debit*. The **BitSwap Ledger** *(one for each connected peer, stores the NodeId of the partner and bytes sent/received)* keeps track of the accounting history of two peers. *Imagining two peers X and Y, Y may send a block to X if X has credit and may not send if it's in debt.*

The **share ratio** defines debt if $0 \leq r < 1$ and is defined as *r = bytes_sent / (bytes_received + 1)* and r it's the parameter of a function that defines the probability of sending a block.

$$P(send/r) = 1 - \frac{1}{(1+e^{(6-3r)})}$$



A sigmoid function

Drops off quickly when r > 2

This ratio's used as a measure of **trust** between two peers, while helping the chocking of free riders too.
Note: *do not penalize too much seeders that want to serve blocks without anything in exchange!*
Note2: *the function implicitly incentives nodes to cache rare pieces even if they're not interested in them!*

To make the BitSwap work, we introduce the next structures:

- **Ledger** *(see earlier)*
- **Need_list:** *multihash of the blocks needed by the current node*
- **Have_list:** *multihash of the blocks that the current node can transfer to the others*
- **Want_list:** *multihash of the blocks wanted by the current node (and peer's peers too!)*

Operations in the BitSwap Protocol

Peers will **open** a connection with a node, they will exchange their ledgers *(filled with 0 if it's the first time)*, then they compare the received ledger with its own ledger and if they match -> **accept**, otherwise refuse.

Send(want_list) is the executed command not only when a connection is opened, but also periodically and whenever the want list is modified. Receivers check if any of the requested blocks are in their **have_list** and depending on the share ratio may transfer data or not.

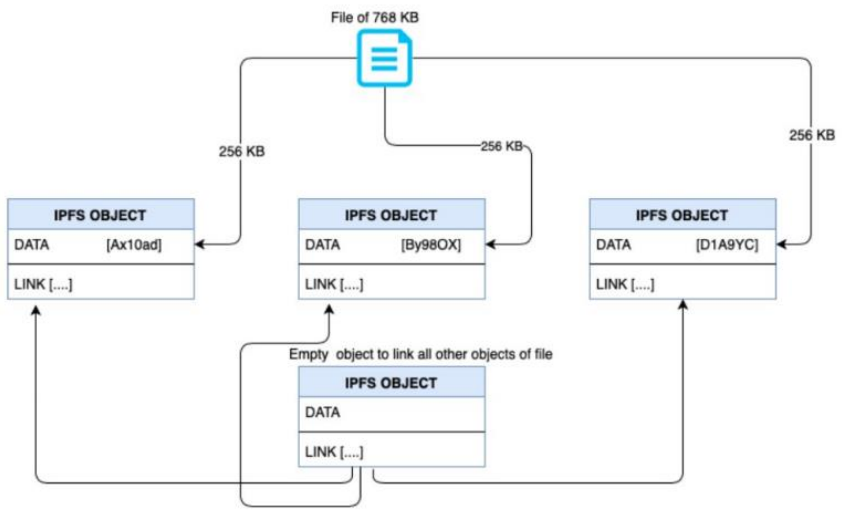
In case they will, **send_block(B)** is invoked, using the protocol specified with the multiaddr of the partner. The receiver will verify the integrity of the data and if it's ok it will remove that block both from *need* and *want list,* while also updating both parts' ledgers.

Useful commands: ipfs bitswap wantlist, ipfs bitswap ledger <nodeId> [shows debt ratio, #exhanges, bytes]

Merkle-DAG

The Merkle-DAG *(Direct Acyclic Graph, a type of graph in which edges have direction and cycles are not allowed)* spec has been deprecated in favor of IPLD [https://docs.ipfs.io/guides/concepts/merkle-dag/]

➔ **Interplanetary Linked Data (IPLD)**



Command: *ipfs object get < … >*

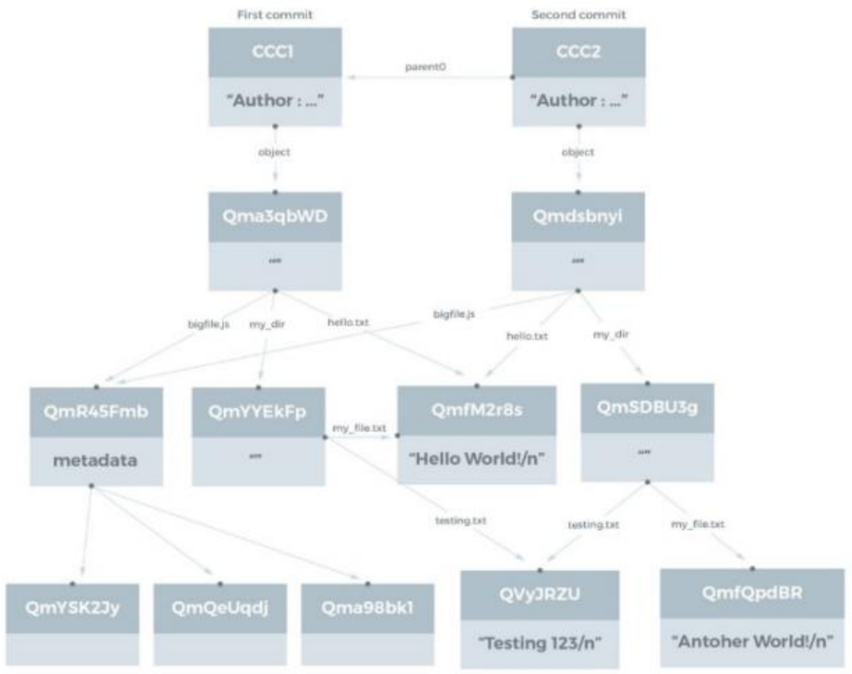
Data: *blob of unstructured binary data (< 256 kB)*

Link: *array of link-structures (containing Name of the link, Hash of the linked object, Size of the whole object, including links)*

We can see it's very similar to Git (see *Distributed Version Control System*)

Main components:

- **Blob**: *content of a file, like an image, video, source code, ..)*
- **Tree:** *content of a directory*
- **Commit:** *information about changes. Points to one tree and 0 or more parents*



But links are hard to read and immutable, so even for different versions of the same content we'd have problems.

IPNS (Interplanetary Naming System)

A decentralized version of DNS, to generate mutable links that point to the latest version of the content and are human-readable *(so, they're easy to remember)*.

How?

By using the DHT to publish the bindings between the stable "Link" and the latest version of an object. It assigns to every peer a mutable namespace, accessed through a permanent link *(/ipns/<nodeId>)* where it can publish links to new versions.

To improve readability:

- mutable links start with */ipns/* and represent pointers to permanent object referred through peerID
  Note: peerID is not human-readable **->** a distributed naming system to translate
- immutable links start with */ipfs/* and represent the actual CID of objects.