The Mainline DHT

Provide a "trackerless" peer discovery mechanism to locate peers belonging to a swarm *(ask to the DHT, not the tracker!)*.

Each node of the DHT stores a part of the tracker information- Now each peer implements both a client/server listening on a TCP port that implements the BitTorrent wire protocol and a client/server listening on an UDP port implementing the DHT protocol. The DHT stores the content's infohash *(key)* and the list of the peers in the swam *(value)*.

Protocol msgs:

- **PING:** probe a node's availability or announce one's existence. If answer fails, purge out of rout. table.
- **GET_PEERS(H):** look for peer belonging to the swarm for the content with infohash *H*. The receiver may store peers for *H (PEER msg)* or have no info about it *(reply with 8 peers closest to H)*
- **ANNOUNCE_PEER:** a peer announces it belongs to a swarm
- **FIND_PEER(id):** request for nodes which are close to the node with node *id*

Routing Table Management

BitTorrent starts with only one bucket, when it's full it's either split or old nodes are pinged *(like Kademlia)*. This implies less memory used and that there's no need to retrieve additional nodes from adjacent buckets.

Each node N in the routing tables of the owner O is characterized by a state, depending on its detected activity in the last interval of times. Its state can be:

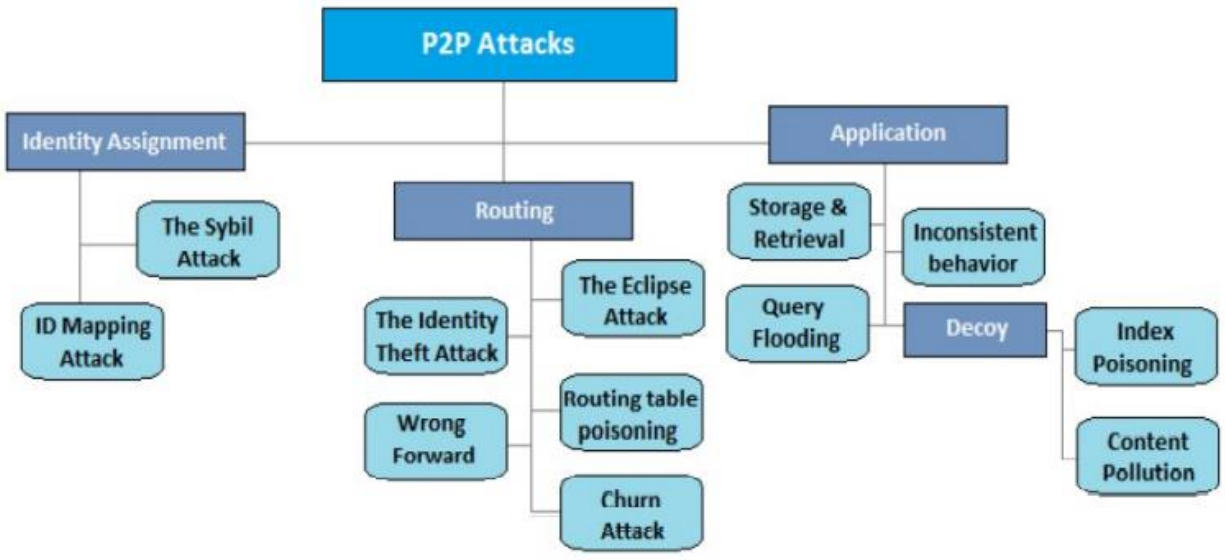
- **GOOD:** N has {queried O} / {responded to one of the queries from O} in the last 15 minutes
- **QUESTIONABLE:** 15 minutes of inactivity
- **BAD:** if the node fails to reply to several queries in a row

If there are *questionable* nodes and a new node wants to be inserted:

1. Send a PING message to each QUESTIONABLE node
2. Who answers becomes GOOD
3. If every node is now good, discard the new node
4. Else, try two consecutive PINGS and, if both fail, replace that node with the new one

```
addNode(Node n_new)
    def Bucket b_old =  bucket where the new node n_new falls
    if ( b_old  is not full ) then
            add n_new to b_old ;
    else
     if (b_old contains myID ) then
        divide b_old , creating two new buckets b_inf and b_sup;
        distribute nodes in b_old and the new node n_new to b_inf e b_sup ;
    else
      if ( all nodes in b_old are GOOD) then
          discard n_new;
      if ( b_old includes a node n  BAD ) then
          substitute n_new to n
      If ( b_old includes nodes QUESTIONABLE) then manageQuestionableNodes(n_new,b_old);
```

P2P Attacks



In a Client-Server approach, the server enforces security, but in P2P there's a lack of the *Trusted Third Party* (TTP).

Identity Assignment

Every entity *(peer or content)* has a unique identifier generated by a cryptographic hash. Some attacks:

- **Sybil atk:** obtain multiple identities. *Injecting one or multiple fake identities (sybils) into the network and using them to perform further attacks like routing attacks, control data replica, win votes by the majority.*

  Case study (Kademlia): first crawl the "zone" to learn about the peers to attack, then attack them by sending PING messages to "poison" their routing tables with pointers to sybils. The peers receiving the message may add them into their routing tables if these are not filled, while the sybils will answer to FIND_NODE(id) with a set of sybils identifiers, such that the requester will have the impression of approaching to the target.

- **ID Mapping atk:** obtain a specific ID, next to the value of the key of a resource you want to control. *Assuming a completely distributed scenario (no central authority who assigns identifiers), the goal is to obtain one set of a particular identifier, in order to gain control over nearby resources. Usually, the node generates its ID "at random" by hashing its IP address and a self-generated public key, which makes this attack harder but doesn't prevent it.*

  Case study (Kademlia): ID space of M bits -> 2^M possible identifiers, N nodes in the network, redundancy factor K (k nodes responsible per resource). The attacker generates random identifiers until the ones he gets are suitable. Probability of success of one try = 1/(N+1), so K successes = K*(N+1). *The complexity doesn't depend on M and grows linearly with K*

Routing

- **Routing Table poisoning:** propagate wrong information for routing. *Peers receive routing updates when they join or someone else joins the overlay and periodically to patch holes and reduce hop delays. The attacker supplies routing updates pointing to controlled nodes. This exploits the fact that it's hard to determine whether routing updates are legitimate or not.* ***Note:*** *bad routing updates are propagated!*

- **Eclipse atk:** Strictly related to sybil and a routing table poisoning atk. *If an attacker controls a sufficient fraction of the neighbors of a current node, it can "eclipse" some correct nodes and all requests will be routed across the attacker, whos then able to provide fake answers or drop messages.*

- **Misroute messages:** change target while forwarding *(randomized or some criteria)*, pretend to be the key manager. *A request is forwarded from a faulty node to a different destination (farther one), implying possible loops or network clogging.*

Application

- **Index Pollution:** observe the hash of the resource on the network, send fake <key, value> correspondences to direct the traffic to a resource R to an invalid destination.

Other attacks are easily explainable by their names.

In Kademlia, we have:

- Iterative look-ups *that allow the querier to check every step of the lookup operation.*
- Redundancy of the resources *that reduces the impact of the DoS attacks or resource hiding*
- Resource expiration and republishing *that minimizes the impact of deliberate churn*

But

- Lacks a mechanism to assure the association between IP address and Kademlia ID
- Lacks a robust ID generation mechanism

Because of this, we introduce the following

Cryptographic Toolbox for DHT and Blockchains

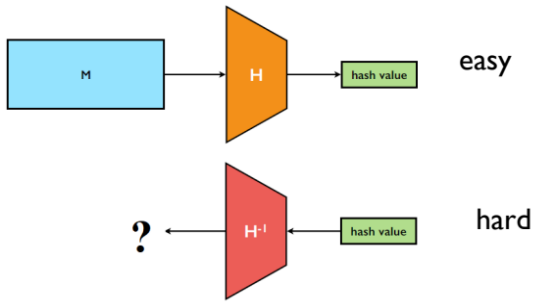
Cryptographic Hash Functions

Also referred as *one-way transformations*, they can take any byte sequence as input and produce a fixed size output *(usually 128, 160, 256, 512 bits)*. They're also efficiently computable and guarantee some security properties, like a small change in the input produces a completely different output, plus classic ones like:

- ***pre-image resistance***

Let X be the domain and Y the codomain of the hash function:

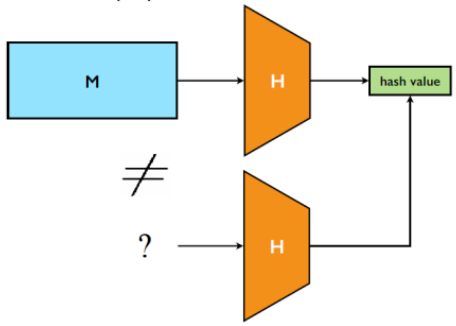
- preimage resistance: for any $y \in Y$, it is hard to find $x \in X$ such that $h(x) = y$
  - one-way function
  - may be also replaced by the hiding property (to be seen later)



- ***second pre-image resistance*** *(weak collision resistance)*

  Let X be the domain and Y the codomain of the hash function:

  - second preimage resistance: given M and thus h = H(M), it is hard to find another value M' that H(M') = h

  - also called weak collision resistance
  - may require exhaustive search looking for M'

  An example: *in the 8-bit block parity, inverting any even number of bits in the same column would not change the parity*

- ***collision-resistance*** *(strong collision resistance)*

  As usual, in any hash function we have collisions and due to the "*Pigeon principle*" we have the codomain that it's smaller of the domain, so we must guarantee that finding these collisions is very hard.

Plus required ones like:

- ***hiding***

  Main problem is that the input domain is easily enumerable: hiding is related to the pre-image property, hide the input of the hashing function. Guaranteeing high min-entropy *(all the values in the distribution are negligibly likely, and no particular value is more likely than others)*. A solution is to pick a random integer of i.e. 256 bits from a distribution with h.m.e., append R to the original input so that the input space becomes extremely hard to enumerate.

  "*Given Hash(secret R || x), it's infeasible to find x.*"

  Note: *in a distributed environment, this implies Commitment! You commit a value, "sealing it", and reveal it later (imagine a game of rock-paper-scissors where someone throws its choice before the other one, he will surely lose!).* Scheme: seal the envelope *commit(secret, nonce)*, *publish* it, *verify* to check validity. As soon as you verify the "game", you know that you have won or not but you're not able to cheat because you'd need to find a collision in order to change the answer, which is hard! *(binding)*

- ***puzzle-friendliness*** *(very useful for cryptocurrencies).*

  This 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: $m = r||x$ $H(m) \in S$

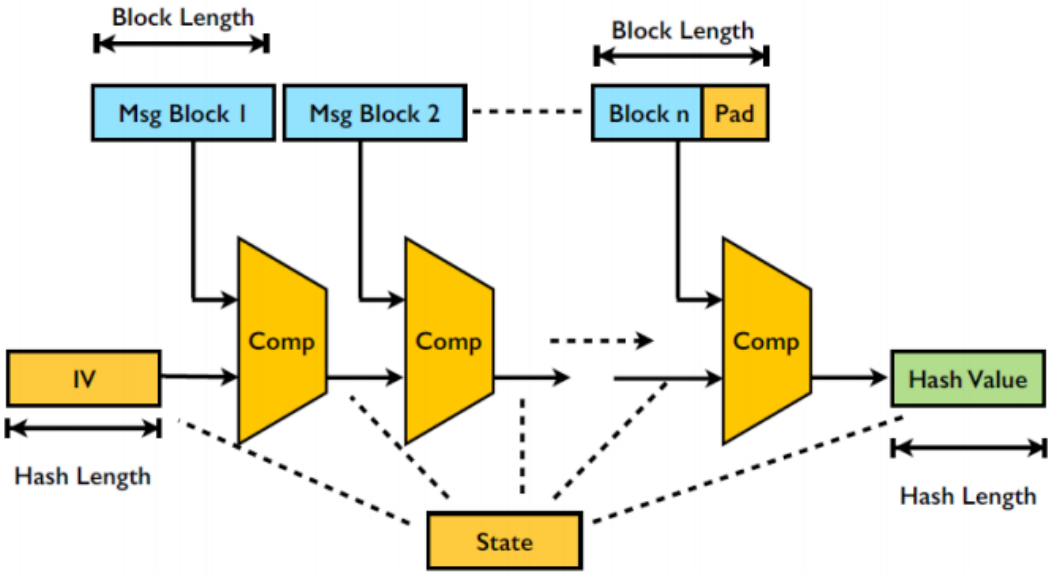
Based on a partial pre-image attack: you have to find a part of the input such that the output belongs to a set (this time not a single value). -> **Bitcoin Proof of Work (PoW)**

Note: *the bigger S is, the easier the puzzle is. In Bitcoin, it's defined by the number of leading zeros of SHA-256.*

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 HME, then it is infeasible to find x such that H(k || x) = y in time significantly less than 2^n. *(it means: no solving strategy to solve a search puzzle is better than trying exhaustively)*

Important mention: **Merckle Damgard Transform**

Used to convert a fixed-length hash function to a hash function taking inputs of arbitrary length, while preserving collision resistance. It's adopted by the most popular hashing functions.



A hash function may be attacked by either exploiting logical weaknesses in the algorithm or by performing a brute-force attack *(exhaustive search)*. The security of a hash code of length n is proportional to:

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

Note: for collision-related things, be careful of the Birthday Paradox *(reminder: n = 23)*

"if every computer ever made by humanity was computing since the beginning of the entire universe, up to now, the probability that they would have found a collision is still infinitesimally small. [narayanan2016bitcoin]"

Hash functions are used to:

- generate data fingerprinting. So if we know that H(x) = H(y), we assume that x = y. Take into account that comparing hashes is definitely faster than comparing entire files!
- Guarantee msg integrity (antitampering): use the hash value as the checksum to check if the data is changed or modified.

Digital signatures

They are the second cryptographic primitive needed as building blocks for blockchains, based on public-key algorithms *(asymmetric).* It does not guarantee data confidentiality, but it ensures integrity. In order to reach confidentiality too, we just encrypt using the previous schemes!

But this is somewhat of a "weak authentication", and it's not sufficient. We need digital certificates and certification authorities

API FOR DIGITAL SIGNATURES

```
(sk, pk) := generateKeys(keysize)
        sk: secret signing key
        pk: public verification key

sig := sign(sk, message) /*cipher the message through the secret
                           key and obtain the signature.

isValid := verify(pk, message, sig) /*decipher the signature
                                      through the public key and
                                   compare the result with the message
```

and the following property must hold:

```
verify(pk, message, sign(sk, message)) == true
```