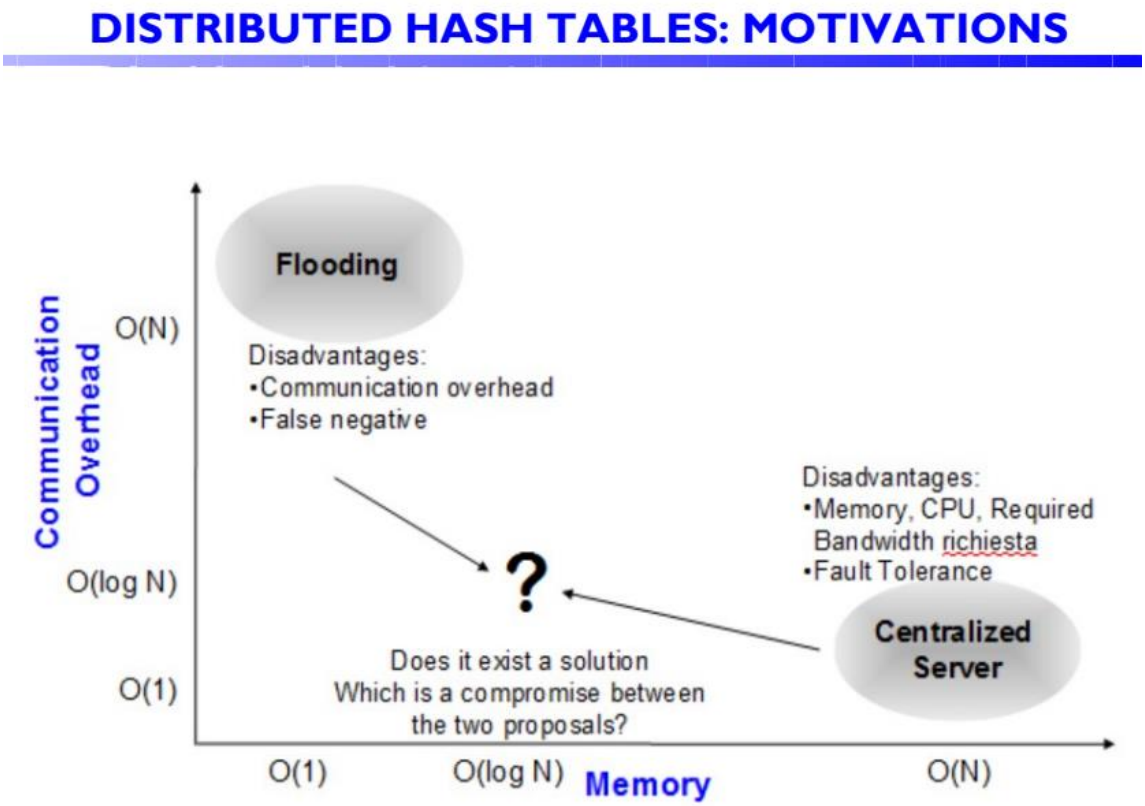
Distributed Hash Tables

Imagine that a peer *A* stores an information *I* and *B* wants to find it.

Where should the information be stored? And how to find it without a centralized server, while still considering system scalability and system adaptability?

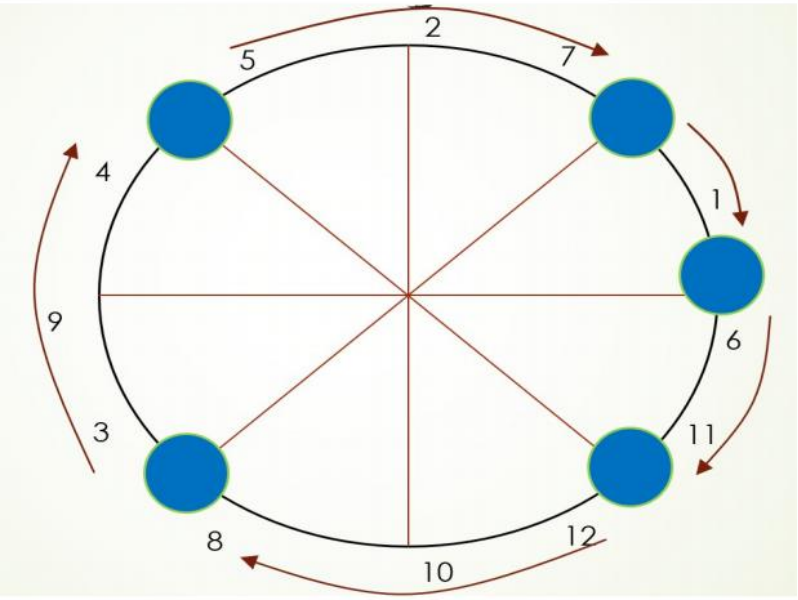
In a pure P2P system, the **search** is guided by the value of a set of attributes, which doesn't require ID computations nor auxiliary structures, but has poor scalability and overhead due to the comparing of whole objects; the **addressing** is the association of a unique identifier to the content and exploited to search.



A **Distributed Hash Table (DHT)** is a distribute set of buckets to peers. To find the peer responsible of a specific bucket, you hash the key, guaranteeing a balance load across the peers. The problem that arises with the usual view of the hash tables is the one of consistent hashing: we cannot use the same mechanisms because the number of peers is very dynamic. *It can be proven that with 10 buckets and 1000 keys nearly 99% of the keys would be remapped constantly.*

So we need to guarantee that adding/removing nodes implies moving only a minority of data items. The basic idea is to let each node manage an interval of consecutive hash keys, not a set of sparse keys.

How? To map an interval to a node, in addition to hashing the names of the objects, hash also the names of all the nodes in the same space.



So, DHT is a distributed hash table realized with consistent hashing, implementing *content addressing*.

Location Addressing **Content Addressing**

- *Classic http link, points to a single location* *- Identify content by its "fingerprint" (hash)*

Having a fingerprint of a content allows you to get the content from anyone who has a copy. This approach is adopted by *Internet Planetary File System (IPFS).*

Identifiers of the peers/data are generate through a cryptographic hash function, usually *SHA-1…512*. But to retrieve the content, the hash function isn't enough!

*Given x, compute h(x) and then **scan** to find the peer p that minimizes: h(p) : h(p) > h(x)*

The cost of the previous lookup depends on the overlay: having huge routing tables would mean O(1) but it's obviously not always possible, so we need routing algorithms.

Content-based Routing: each node maintains a routing table storing a partial view of the network.

A node **leaving** can be for two different reasons:

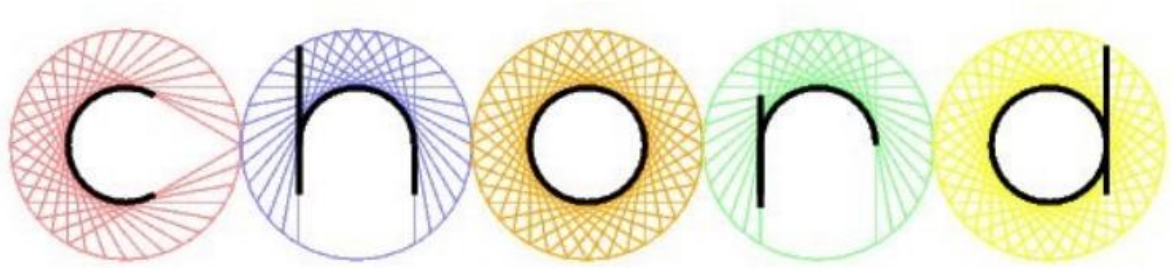
- Voluntarily, we then partition its address space to the neighbours and delete the node from the routing tables
- Node failure, we need some redundancy and periodical information refresh *(some probing too!)*

The **load balancing** is solved by exploiting an *Uniform Hash Function*, guaranteeing robustness and scalability.

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

Chord

Paper reference Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. IEEE/ACM Transactions on Networking developed in 2001 from a research group formed by researcher from MIT and University of California



Main charasteristics:

- applies consistent hashing concept
- routing to find the node managing a key costs log(N) hops *with high probability*
- routing table size is log(N) *with high probability (w.h.p.)*
- self-organization and adaptation to the churn

The API has 4 functions:

- **Join:** join the network
- **Leave:** leave the network
- **Store:** store a key-value pair
- **Retrieve:** retrieve a stored value

Each **node** is paired with identifier ID, obtained by SHA — *idnode = SHA (IP address, port)*

Each **data** is paired with a unique identifier k — *k = SHA (data_name)*

And as we previously stated, both keys and nodes are mapped in the same logical space. The mapping algorithm consists in assigning a key *k* to the first node *n* of the ring whose identifier is greater or equal to K *(clockwise)*.

The look-up depends from the information stored in the routing table -> different tables:

- **Minimal Routing Table:** each node remembers only the next node on the ring
  Implies O(1) space but O(n/2) time

To reduce the number of steps, each node should have more than 1 neighbour. *(z = n -> complete mesh)*

Since 1 and n are bad, we try a compromise: *a logarithmic mesh of the nodes of the ring.* Each node stores several links towards close neighbours and few ones towards fars *(routing is more accurate in the proximity of the node, while more approximate far away)*. This way, the routing algorithm can be resumed as: "send a query for the key *k* to the farthest known predecessor of *k*"

To support routing, each node will also save a link to the successor and predecessor on the ring.

How to guarantee ring's maintenance *(handling churn)*? Have **more than one successor**, so that if one disconnects there are others. This way, the correctness of the protocol depends on the consistency of all the successor pointers!

When a new node connects, it finds his successor by sending a query *findSuccessor()* with the identifier of the node itself as key, it then links to its successor and similarly the successor sets its new predecessor. But the "old predecessor" doesn't know about the new node! -> stabilization procedure

If (successor.getPredecessor() != (null || me)) setSuccessor(successor.getPredecessor());

But finger tables of other nodes become inconsistent too! -> Periodical stabilization of the finger tables.

At each iteration, fix a finger *(or choose one at random) === submit a query with that finger as target*

Then this means that a search initiated before the system is in stable state:

- May fail if not all the pointers to the successor nodes are stable *(false negatives)*
- May be slowed down (no false negatives) due to not completely updated finger tables *(efficiency)*
- May do good *O(log N) steps* when all the finger tables are "reasonably updated"

More robustness <-> bigger successors' list *A good value for list's size is log(N)*

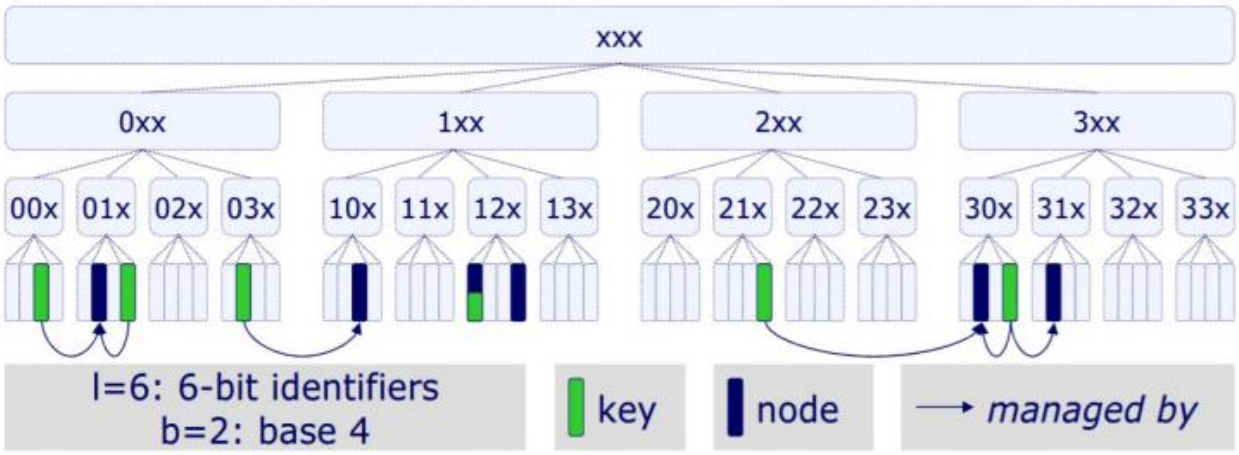
Furthermore, a node crash would leave two dangling references. What happens here is that successor of the crashed node will note the crash *(who knows when)*, sets his predecessor to null, then the "in that moment predecessor" would check its successor and notify it so that it can get the inverse link.

Prefix-Based DHT

Basic idea: a generalization of the routing on hypercubes. Mapping of nodes and keys to numbers *(m digits)* and then assign each key to the node with which it shares the longest prefix, if possible.

Example: *321302, base = 4, m = 6* *A: 321002* *B: 321333* *will choose B*

The identifier space is modelled by a tree, where *l* is the length of the identifiers and *depth(tree) ≤ l.* Each node has *b* sons *(the considered base)*



Uses Plaxton Routing: correct a digit of the considered base at each step.

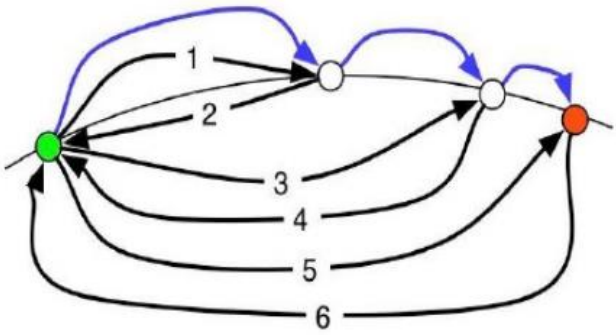
- In base *b*, then b bits are corrected at each step.
- There are a list of references to other nodes called *K-buckets*.
- At each look-up step, each node has the possibility to choose among *K* different contacts. If *K > 1* then we guarantee an higher robustness and tolerance to faults due to the possibility to both choose among alternative routing paths and the parallel search on different paths!

In Kademlia, *b = 1* -> binary tree, *K* ≅ 20.

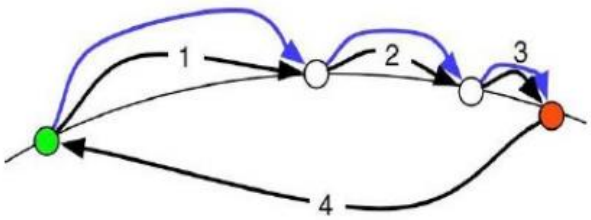
The size of the node's routing table and the number of look-up hops depends on b *O(log_b(n))*

The routing may be iterative or recursive *(see slide):*

- iterative routing:
  - node sending the look up request manages the search process
  - at each routing step, that node waits for a reply
  - the received reply includes a notification of the next routing step



- recursive routing: look up passes from node to node without the intervention of the starting node



- Kademlia
  - iterative routing

Case study: Kademlia

"the de facto standard searching algorithm for P2P networks on the Internet"

A protocol specification for efficiently storing and retrieving data across the network in P2P environments.

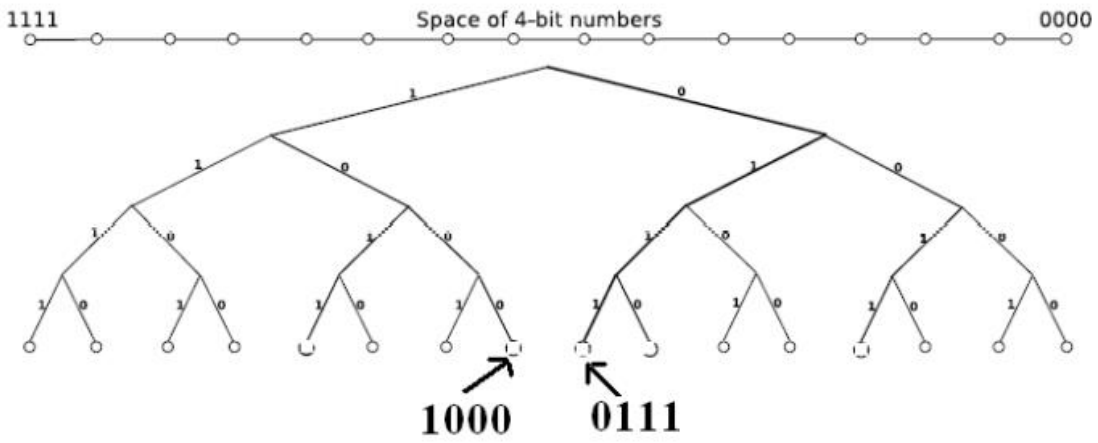
- **Decentralized:** data stored redundantly on peers
- **Fault Tolerant:** the data is being stored on multiple peers, even if one drops out, it'll still be there
- **No need for complex DB engines:** data stored as <key, value>, making it suitable even for IoT tools

It presents a set of charasteristics which are not offered by any existing DHT, such as:

- Routing information spreads automatically as a side-effect of look-ups
- Flexibility to send multiple requests in parallel to speed up look-ups *(parallel, param α)*

Uses the **XOR metric**, where the "distance" between two objects is the result of the XOR operation on their identifiers, interpreted as an unsigned integer. [It's a metric because d(x,y) is always > 0 and == 0 iff x=y, it's equal to d(y,x) *simmetry*, the d(x,y) XOR d(y,z) = d(x,z) *transitivity* and so also $\geq$ d(x,z) *triangular inequality. Furthermore it is unidirectional!*]

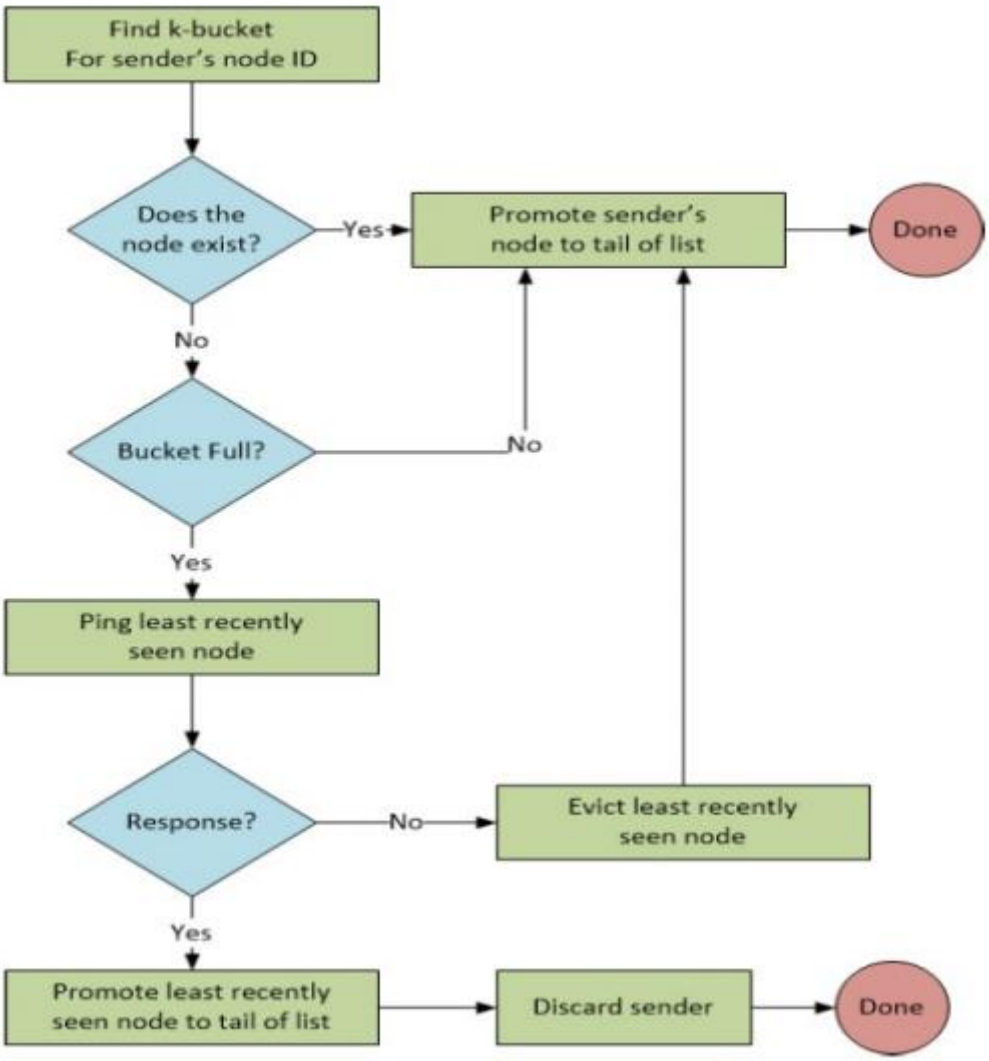
Due to this metric, "close" nodes are characterized by a long common prefix.



- the identifier space of Kademlia is a binary balanced tree
- two leaves may be close in the tree and numerically close, but they are distant according to the metrics ⊕

The routing table's rows are k-buckets, each one containing k contacts *(== each corresponds to a subtree)*. The first entries correspond to peers sharing a long prefix, while the last ones are "farther".

To manage the K-buckets, we simply follow this schema:



We prefer old contacts with respect to newer ones because, due to Gnutella's experience, the longer a node has been up, the more likely it is to remain up another hour. A benefir of the K-buckets is the resistance to DoS attacks *(can't flood system with new nodes, they will be inserted only when old nodes leave the system)*.

To keep the K-bucket up to date, they are refreshed for each query passing through the node AND periodically *(once each hour)*.

Note: *the dispatch of the query to the node which is closest to the target doesn't imply that it's the smaller path towards the target!*

The unidirectionality of the XOR metric guarantees that all the paths converge towards the target!

Basic protocol operations *(through Remote Procedure Calls)***:**

- **FIND_VALUE:** *if the value is present then it's returned, otherwise it'll be a FIND_NODE*
- **PING:** *probe the receiver node to see if it's online*
- **STORE:** *instructs the receiver node to store a <key, value> pair*
- **FIND_NODE:** *returns <IP_address, UDP_port, Node_ID> for the k known nodes closest to the target.*

To store a <k, v> pair, a participant performs a look-up to find the k closest nodes and sends them STORE operations. Then either some of the k nodes (or all) leave the network or new nodes enter the networks with ID closer to some already published keys -> **re-publishing mechanism** *(each node republished <k,v> pairs as necessary to keep them alive)*. In Kademlia, every 24h.

The join is simply the FIND_NODE to the bootstrap node in order to find other nodes close to itself.

KADEMLIA: SUMMARY

Strengths

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

Weaknesses

- non-uniform distribution of nodes in ID-space results into imbalanced routing table and inefficient routing
- balancing of storage load is not truly solved
- originally underspecified, plethora of different implementations
- hard to provide analytical results
- non-deterministic results of routing (time, neighbourhood)