Large-Scale Data Systems

Lecture 9: Distributed Hash Tables

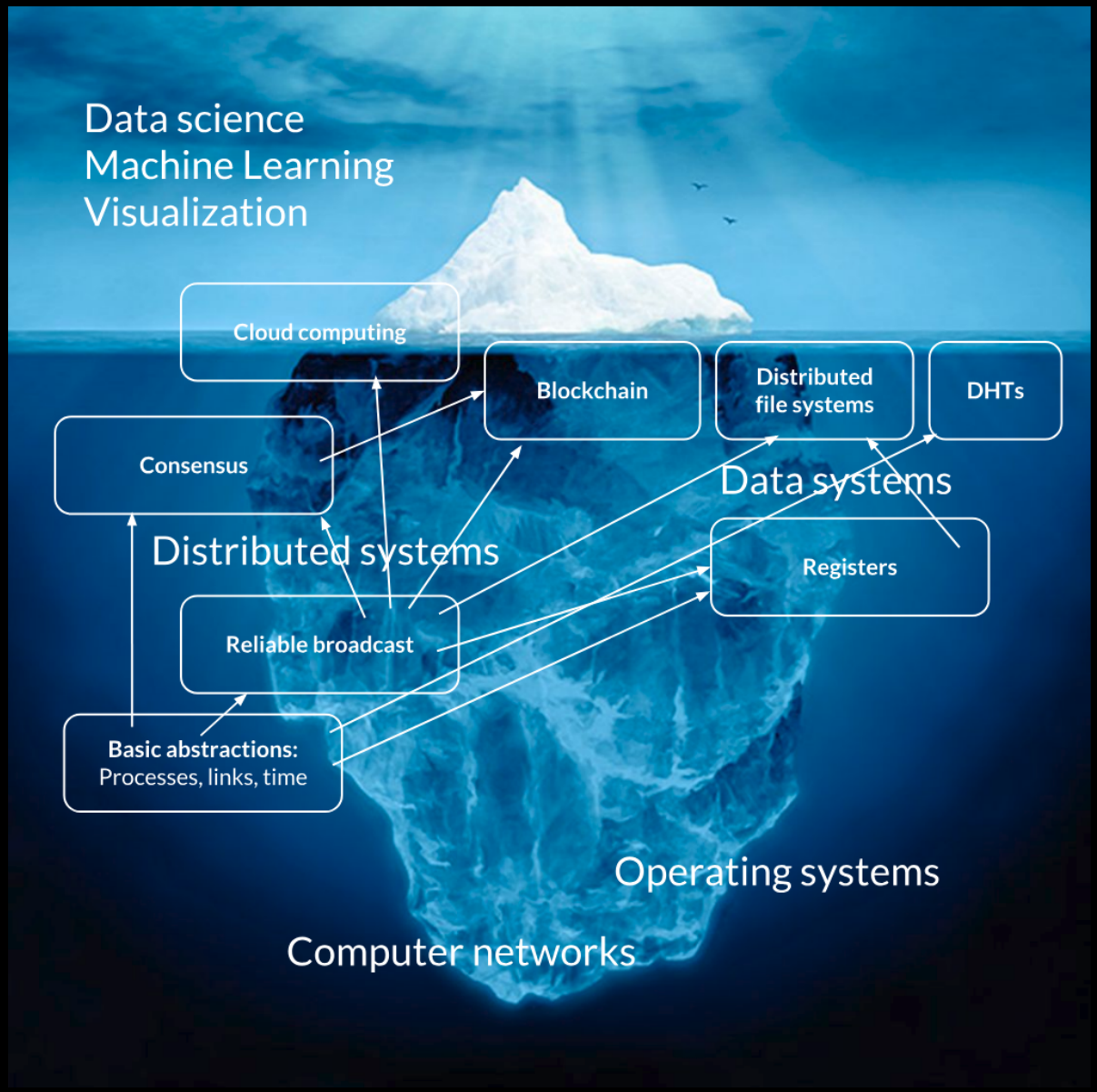
Prof. Gilles Louppe
g.louppe@uliege.be

LIÈGE université

Today

How to design a large-scale distributed system similar to a hash table?

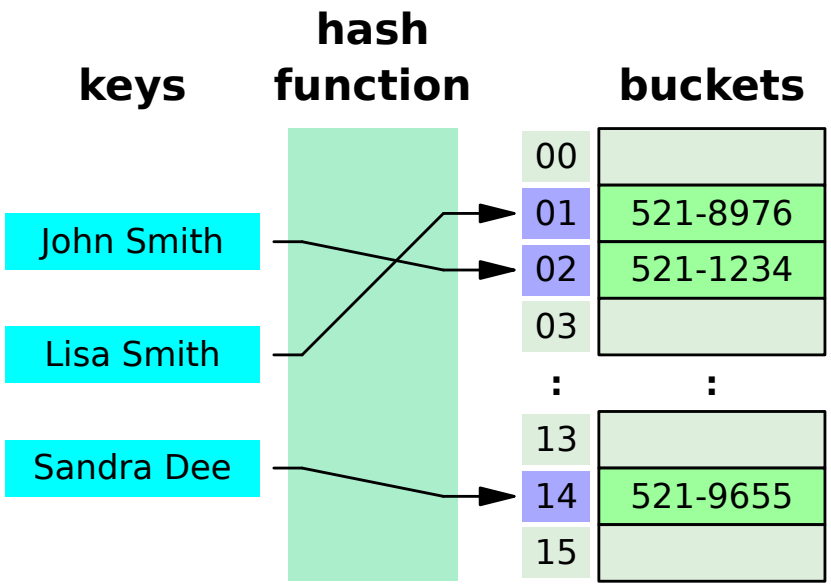
- Chord
- Kademlia


Data science
Machine Learning
Visualization
Cloud computing
Blockchain
Distributed file systems
DHTs
Consensus
Data systems
Distributed systems
Registers
Reliable broadcast
Basic abstractions:
Processes, links, time
Operating systems
Computer networks

Hash tables

A hash table is a data structure that implements an associative array abstract data type, i.e. a structure than can map keys to values.

- It uses a hash function to compute an index into array of buckets or slots, from which the desired value can be found.
- Efficient and scalable: $\mathcal{O}(1)$ look-up and store operations (on a single machine).



Distributed hash tables

A distributed hash table (DHT) is a class of decentralized distributed systems that provide a lookup service similar to a hash table.

- Extends upon multiple machines in the case when the data is so large we cannot store it on a single machine.
- Robust to faults.

Interface

- $\mathrm{put}(k, v)$
- $\mathrm{get}(k)$

Properties

- When $\mathrm{put}(k, v)$ is completed, k and v are reliably stored on the DHT.
- If k is stored on the DHT, a process will eventually find a node which stores k.

Chord

Chord

Chord is a protocol and algorithm for a peer-to-peer distributed hash table.

- It organizes the participating nodes in an overlay network, where each node is responsible for a set of keys.
- Keys are defined as m-bit identifiers, where m is a predefined system parameter.
- The overlay network is arranged in a identifier circle ranging from 0 to $2^m - 1$.
  - A node identifier is chosen by hashing the IP address.
  - A key identifier is chosen by hashing the key.
- Based on consistent hashing with SHA-1 hash function.
- Supports a single operation: $\text{lookup}(k)$.
  - Returns the host which holds the data associated with the key.

Consistent hashing

Traditional hashing

- Set of n bins.
- Key k is assigned to a particular bin.
- If n changes, all items need to be rehashed.
  - E.g. when `bin_id = hash(key) % num_bins`.

Consistent hashing

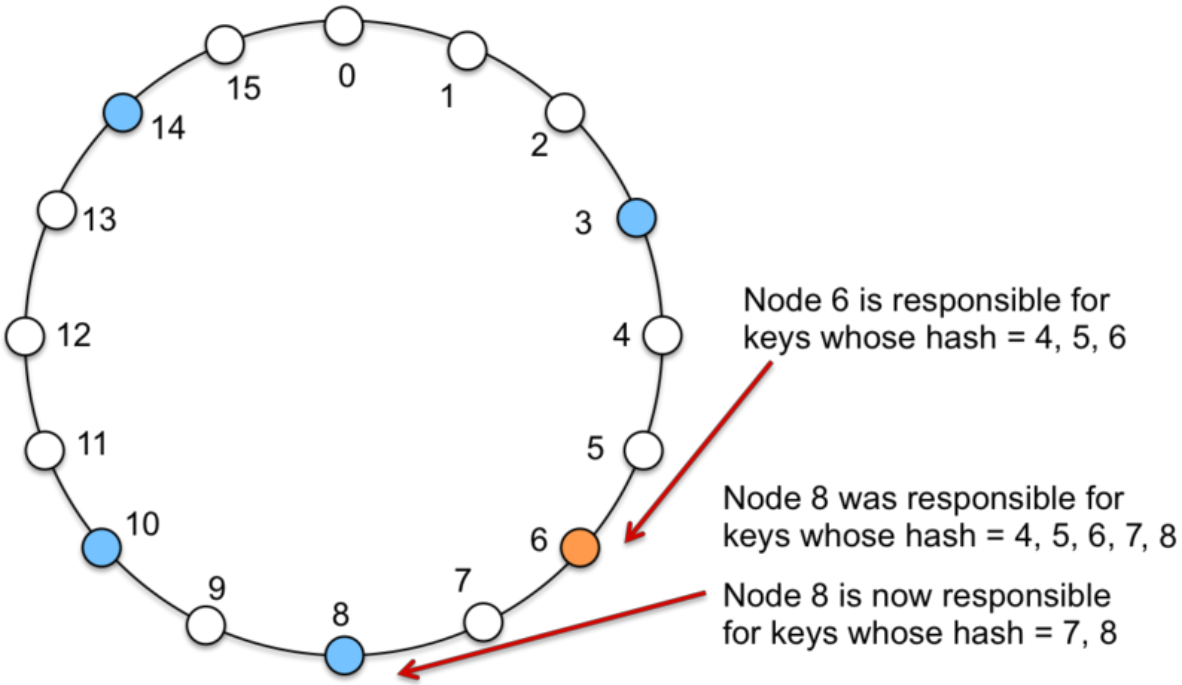
- Evenly distributes x objects over n bins.
- When n changes:
  - Only $\mathcal{O}\left(\frac{x}{n}\right)$ objects need to be rehashed.
  - Uses a deterministic hash function, independent of n.

Consistent hashing in Chord assigns keys to nodes as follows:

- Key k is assigned to the first node whose identifier is equal to or follows k in the identifier space.
  - i.e., the first node on the identifier ring starting from k.
- This node is called the successor node of k, denoted $\text{successor}(k)$.
- Enable minimal disruption.

To maintain the consistent (hashing) mapping, let us consider a node n which

- joins: some of the keys assigned to $\text{successor}(n)$ are now assigned to n.
  - Which? $\text{predecessor}(n) < k \leq n$
- leaves: All of n's assigned keys are assigned to $\text{successor}(n)$.



Routing

The core usage of the Chord protocol is to query a key from a client (generally a node as well), i.e. to find $\text{successor}(k)$.

Basic query

- Any node n stores its immediate successor $\text{successor}(n)$.
- If the key cannot be found locally, then the query is passed to the node's successor.
- Scalable, but $\mathcal{O}(n)$ operations are required.
    - Unacceptable in large systems!

Finger table

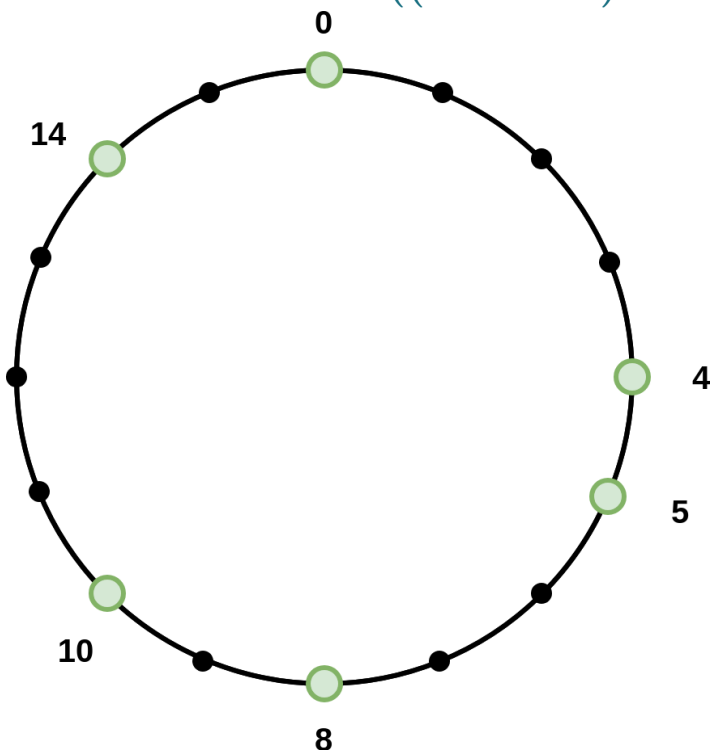
In Chord, in addition to successor and predecessor pointers, each node maintains a finger table to accelerate lookups.

- As before, let m be the number of bits in the identifier.
- Every node n maintains a routing (finger) table with at most m entries.
- Entry i in the finger table of node n:
  - First node s that succeeds n by at least 2^{i-1} on the identifier circle.
  - Therefore, $s = \text{successor}((n + 2^{i-1}) \bmod 2^m)$

Notation	Definition
$finger[k].start$	$(n + 2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
$.interval$	$[finger[k].start, finger[k+1].start)$
$.node$	first node $\geq n.finger[k].start$
successor	the next node on the identifier circle; $finger[1].node$
predecessor	the previous node on the identifier circle

Finger table example

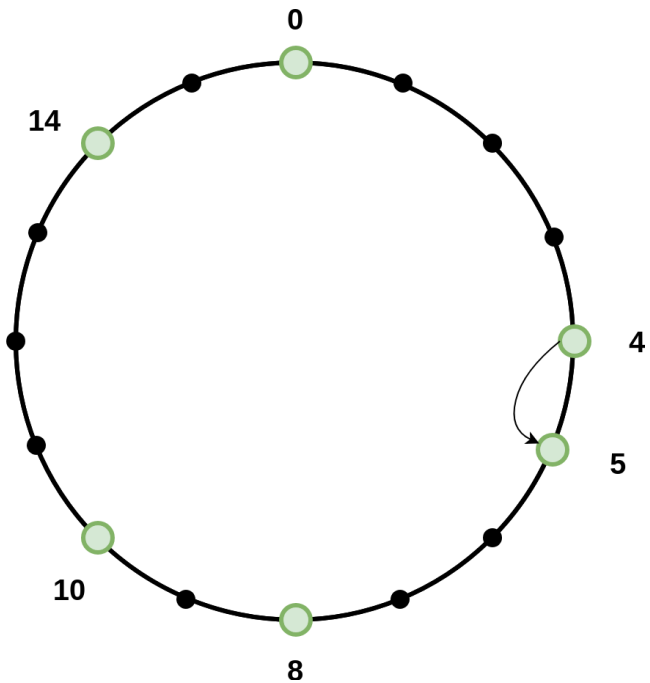
- $m = 4$ bits $\rightarrow$ max 4 entries in the routing table.
- i-th entry in finger table: $s = \text{successor}((n + 2^{i-1}) \bmod 2^m)$



First entry

- $n = 4, i = 1$
- $s = \text{successor}((n + 2^{i-1}) \bmod 2^m) = \text{successor}(5) = 5$

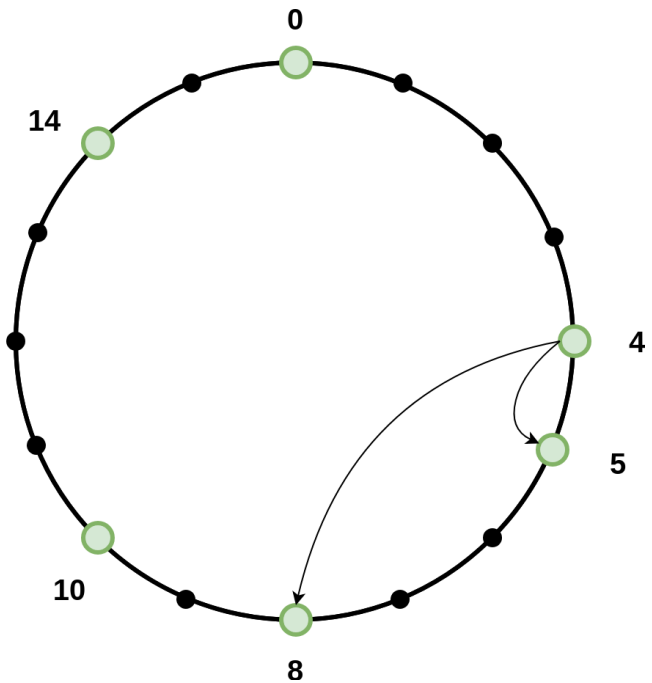
Entry	Interval	Successor
1	[5,6)	5



Second entry

- $n = 4, i = 2$
- $s = \text{successor}((n + 2^{i-1}) \bmod 2^m) = \text{successor}(6) = 8$

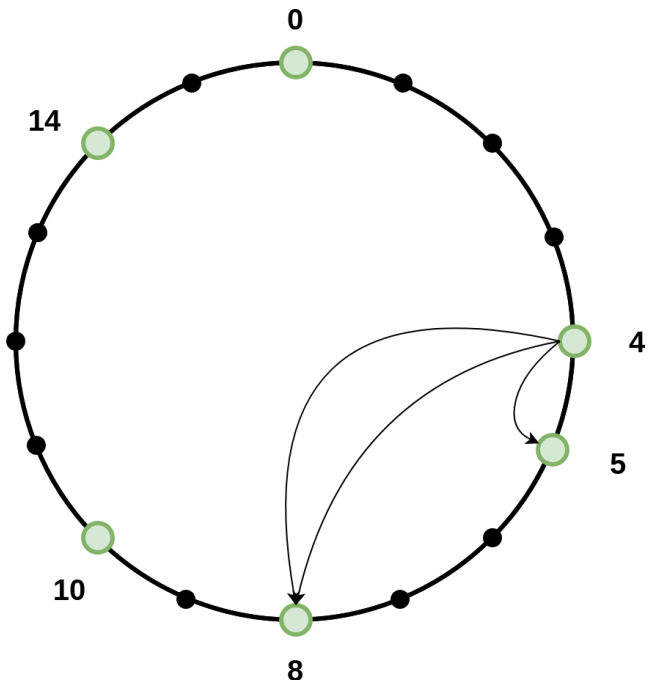
Entry	Interval	Successor
1	[5,6)	5
2	[6,8)	8



Third entry

- $n = 4, i = 3$
- $s = \text{successor}((n + 2^{i-1}) \bmod 2^m) = \text{successor}(8) = 8$

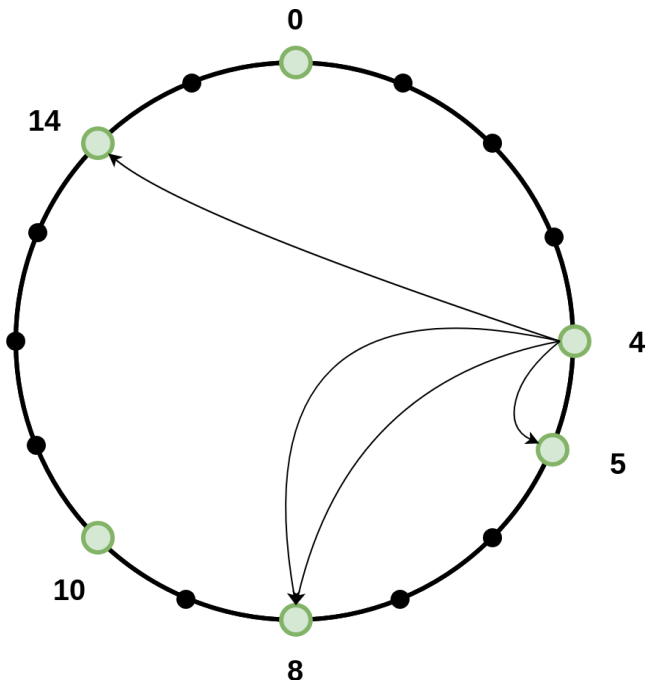
Entry	Interval	Successor
1	[5,6)	5
2	[6,8)	8
3	[8,12)	8



Fourth entry

- $n = 4, i = 4$
- $s = \text{successor}((n + 2^{i-1}) \bmod 2^m) = \text{successor}(12) = 14$

Entry	Interval	Successor
1	[5,6)	5
2	[6,8)	8
3	[8,12)	8
4	[12, 4)	14



Improved lookup

A lookup for $\text{successor}(k)$ now works as follows:

- if k falls between n and $\text{successor}(n)$, return $\text{successor}(n)$.
- otherwise, the lookup is forwarded at n', where n' is the node if the finger table that most immediately precedes k.
- Since each node has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target key.
- $\mathcal{O}(\log N)$ nodes need to be contacted.

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    // forward the query around the circle
    n0 = successor.closest_preceding_node(id);
    return n0.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i = m downto 1
    if (finger[i]∈(n,id))
      return finger[i];
  return n;
```

Example: finding $\text{successor}(k = 3)$ from $n = 4$

1. $n = 4$ checks if k is in the interval (4, 5].
2. It not, $n = 4$ checks its finger table (starting from the last entry, i.e., $i = m$).
    1. Is node 14 in the interval (4, 4)? Yes!
3. $n = 14$ checks if k is in the interval (14, 0].
4. No, $n = 14$ checks its finger table for closest preceding node.
    1. Return node 0.
5. $n = 0$ checks if k is in the interval (0, 4]. Yes!

$\rightarrow$ Node 0 is the preceding node of $k = 4$. Therefore $\text{successor}(k = 3) = 4$.

Of course, one could implement a mechanism that prevents node 4 from looking up its own preceding node in the network.

Join

We must ensure the network remains consistent when a node n joins by connecting to a node n'. This is performed in three steps:

1. Initialize the successor of n.
2. Update the fingers and predecessors of existing nodes to reflect the addition of n.
3. Transfer the keys and their corresponding values to n.

Initializing n's successor

n learns its successor by asking n' to look them up $\text{find-predecessor}(n)$.

```
// join a Chord ring containing node n'.
n.join(n')
  predecessor = nil;
  successor = n'.find_successor(n);
```

Periodic consistency check

```
// called periodically. n asks the successor
// about its predecessor, verifies if n's immediate
// successor is consistent, and tells the successor about n
n.stabilize()
  x = successor.predecessor;
  if (x∈(n, successor))
    successor = x;
  successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
  if (predecessor is nil or n'∈(predecessor, n))
    predecessor = n';

// called periodically. refreshes finger table entries.
// next stores the index of the finger to fix
n.fix_fingers()
  next = next + 1;
  if (next > m)
    next = 1;
  finger[next] = find_successor(n+2^{next-1});
```

Transferring keys

- n can become the successor only for keys that were previously the responsibility of the node immediately following n.
- n only needs to contact the successor of $n + 1$ to transfer responsibility of all relevant keys.

Fault-tolerance

Failures

- Since the successor (or predecessor) of a node may disappear from the network (because of failure or departure), each node records a whole segment of the circle adjacent to it, i.e., the r nodes following it.
- This successor list results in a high probability that a node is able to correctly locate its successor (or predecessor), even if the network in question suffers from a high failure rate.

Replication

- Use the same successor-list to replicate the data!

Summary

- Fast lookup $\mathcal{O}(\log N)$, small routing table $\mathcal{O}(\log N)$.
- Handling failures and addressing replication (load balance) using same mechanism (successor list).
- Relatively small join/leave cost.
- Iterative lookup process.
- Timeouts to detect failures.
- No guarantees (with high probability ...).
- Routing tables must be correct.

Kademlia

Kademlia

Kademlia is a peer-to-peer hash table with provable consistency and performance in a fault-prone environment.

- Configuration information spreads automatically as a side-effect of key look-ups (gossiping).
- Nodes have enough knowledge and flexibility to route queries through low-latency paths.
- Asynchronous queries to avoid timeout delays from failed nodes.
- Minimizes the number of configuration messages (guarantee).
- 160-bit identifiers (e.g., using SHA-1 or some other hash function, implementation specific).
- Key-Value pairs are stored on nodes based on closeness in the identifier space.
- Identifier based routing algorithm by imposing a hierarchy (virtual overlay network).

System description

- Treat nodes as leaves in an (unbalanced) binary tree, wich each node's position determined by the shortest unique prefix of its ID.
- The Kademlia protocol ensures that every node knows at least one other node in every sub-tree. This guarantees that any node can locate any other node given its identifier.

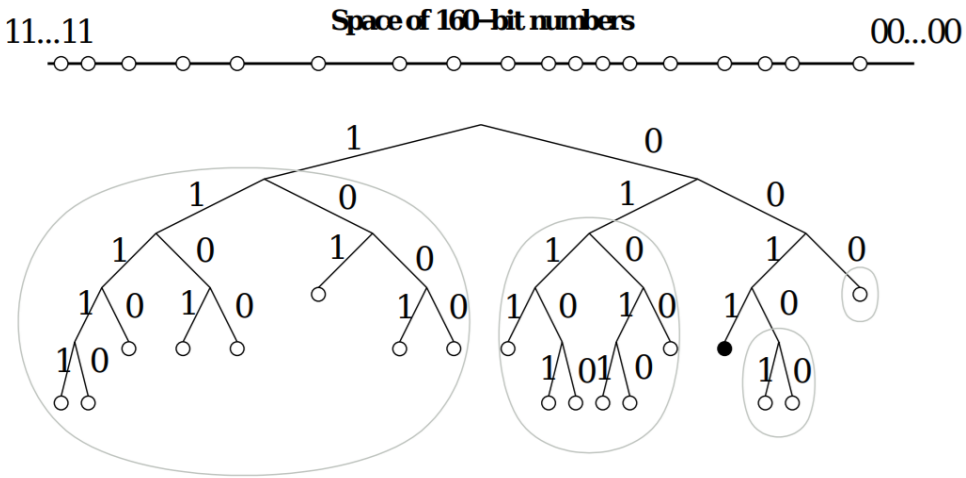


Fig. 1: Kademlia binary tree. The black dot shows the location of node $0011\cdots$ in the tree. Gray ovals show subtrees in which node $0011\cdots$ must have a contact.

Node distance

The distance between two identifiers is defined as

$$d(x, y) = x \oplus y.$$

- XOR is valid, albeit non-Euclidean metric.
- XOR captures the notion of distance between two IDs: in a fully-populated binary tree of 160-bit IDs, it is the height of the smallest subtree containing them both.
- XOR is symmetric.
- XOR is unidirectional.

Node state

- For every prefix $0 < i < 160$, every node keeps a list, called a k-bucket, of (IP address, Port, ID) for nodes of distance between 2^i and 2^{i+1} of itself.
- Every k-bucket is sorted by time last seen (descending, i.e, last-seen first).
- When a node receives a message, it updates the corresponding k-bucket for the sender's identifier. If the sender already exists, it is moved to the tail of the list.
    - Important: If the k-bucket is full, the node pings the last seen node and checks if it is still available. Only if the node is not available it will replace it.
    - Policy of replacement only when a nodes leaves the network $\rightarrow$ prevents Denial of Service (DoS) attacks (e.g., flushing routing tables).

k-bucket

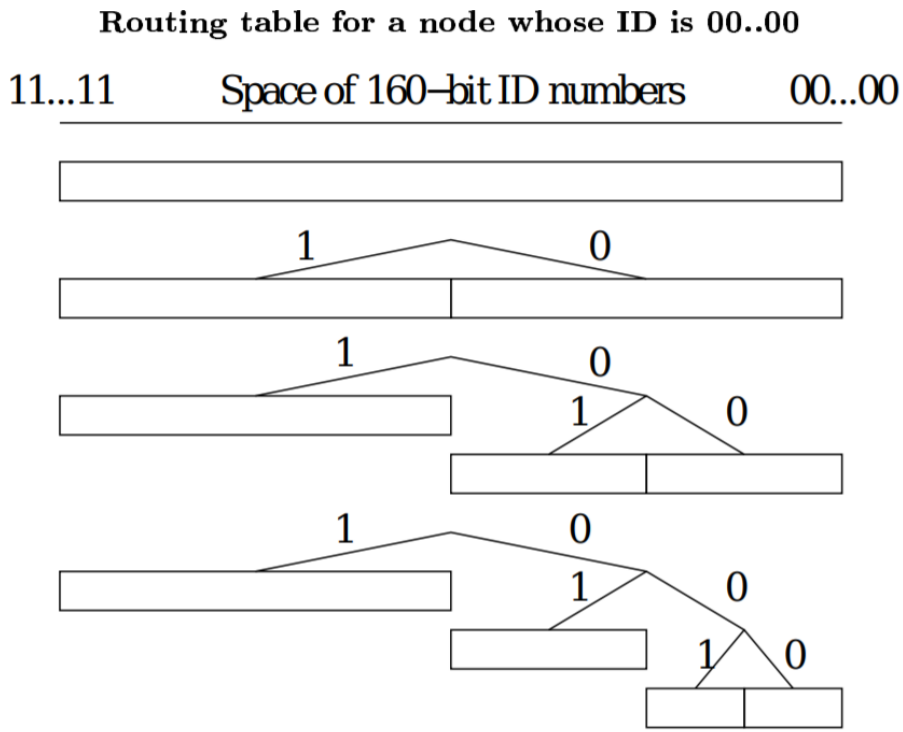


Fig. 4: Evolution of a routing table over time. Initially, a node has a single k-bucket, as shown in the top routing table. As the k-buckets fill, the bucket whose range covers the node's ID repeatedly splits into two k buckets.

Interface

Kademlia provides four remote procedure calls (RPCs):

- `PING(id)` returns (IP, Port, ID)
  - Probes the node to check whether it is still online.
- `STORE(key, value)`
- `FIND_NODE(id)` returns (IP, Port, ID) for the k nodes it knows about closest to ID.
- `FIND_VALUE(key)` returns (IP, Port, ID) for the k nodes it knows about closest to ID, or the value if it maintains the key.

Node lookup

The most important procedure a Kademlia participant must perform is locating the k closest nodes to some given identifier.

- Kademlia achieves this by performing a recursive lookup procedure.
- The initiator issues asynchronous `FIND_NODE` requests to α (system parameter) nodes from its closest non-empty k-bucket.
  - Parallel search with the cost of increased network traffic.
  - Nodes return the k closest nodes to the query ID.
  - Repeat and select the α nodes from the new set of nodes.
  - Terminate when set doesn't change.
  - Possible optimization: choose α nodes with lowest latency.

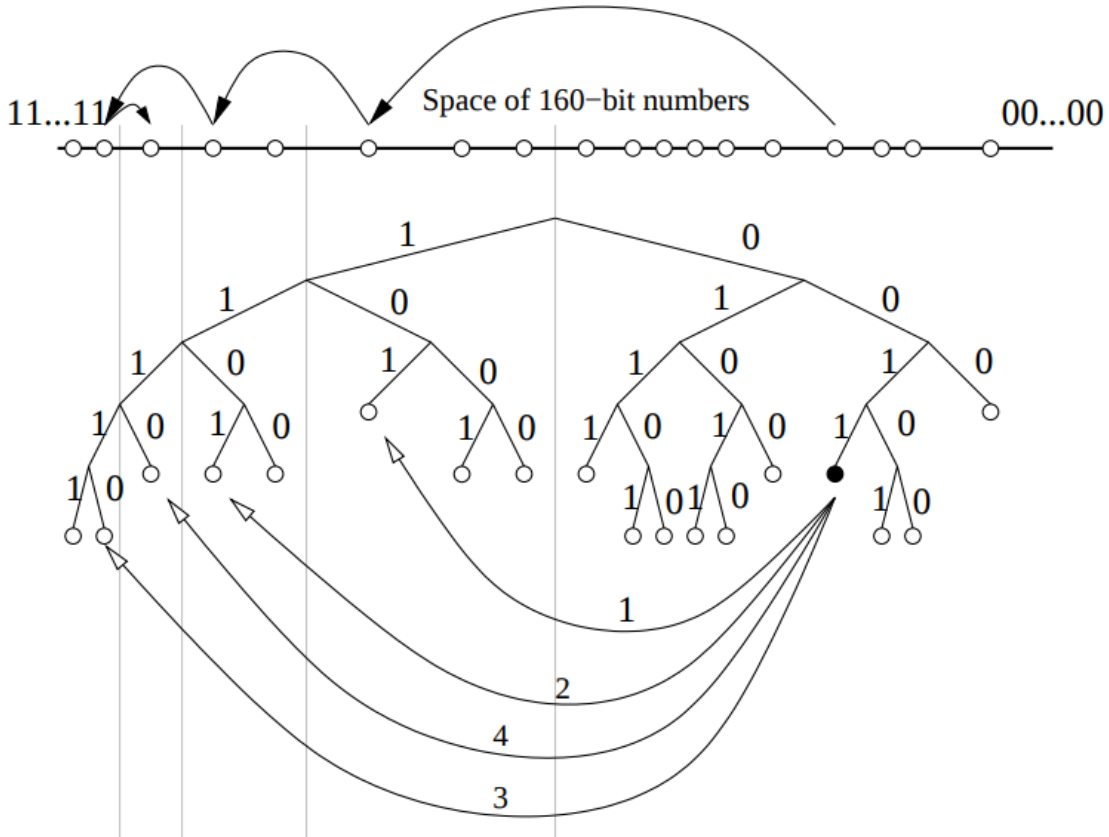


Fig. 2: Locating a node by its ID. Here the node with prefix 0011 finds the node with prefix 1110 by successively learning of and querying closer and closer nodes. The line segment on top represents the space of 160-bit IDs, and shows how the lookups converge to the target node. Below we illustrate RPC messages made by 1110. The first RPC is to node 101, already known to 1110. Subsequent RPCs are to nodes returned by the previous RPC.

Storing data

Using the lookup procedure, storing and making data persistent is trivial.

$\rightarrow$ Send a STORE RPC to the k closest nodes identified by the lookup procedure.

- To ensure persistence in the presence of node failures, every node periodically republishes the key-value pair to the k closest nodes.
- Updating scheme can be implemented. For example: delete data after 24 hours after publication to limit stale information.

Retrieving data

1. Find k closest nodes of the specified identifier using `FIND_VALUE(id)`.
2. Halt procedure immediately whenever the set of closest nodes doesn't change or a value is returned.

$\rightarrow$ For caching purposes, once a lookup succeeds, the requesting node stores the key-value pair at the closest node to the key that did not return the value.

Because of the unidirectionality of the topology (requests will usually follow the same path), future searches for the same key are likely to hit cached entries before querying the closest node.

$\rightarrow$ Induces problem with popular nodes: over-caching.

Solution: Set expiration time inversely proportional to the distance between the true identifier and the current node identifier.

Join

Straightforward approach compared to other implementations.

1. Node n initializes it's k-bucket (empty).
2. A node n connects to an already participating node j.
3. Node n then performs a node-lookup for its own identifier.
    - Yielding the k closest nodes.
    - By doing so n inserts itself in other nodes k-buckets.

Note: The new node should store keys which are the closest to its own identifier by obtaining the k-closest nodes.

Leave and failures

Leaving is very simple as well. Just disconnect.

- Failure handling is implicit in Kademlia due to data persistence.
- No special actions required by other nodes (failed node will just be removed from the k-bucket).

Routing table

The routing table is an (unbalanced) binary tree whose leaves are k-buckets.

- Every k-bucket contains some nodes with a common prefix.
- The shared prefix is the k-buckets position in the binary tree.
- Thus, a k-buckets covers some range of the 160 bit identifier space.
- All k-buckets cover the complete identifier space with no overlap.

Dynamic construction of the routing table

- Nodes in the routing table are allocated dynamically as needed.
- A bucket is split whenever the k-bucket is full and the range includes the node's own identifier.

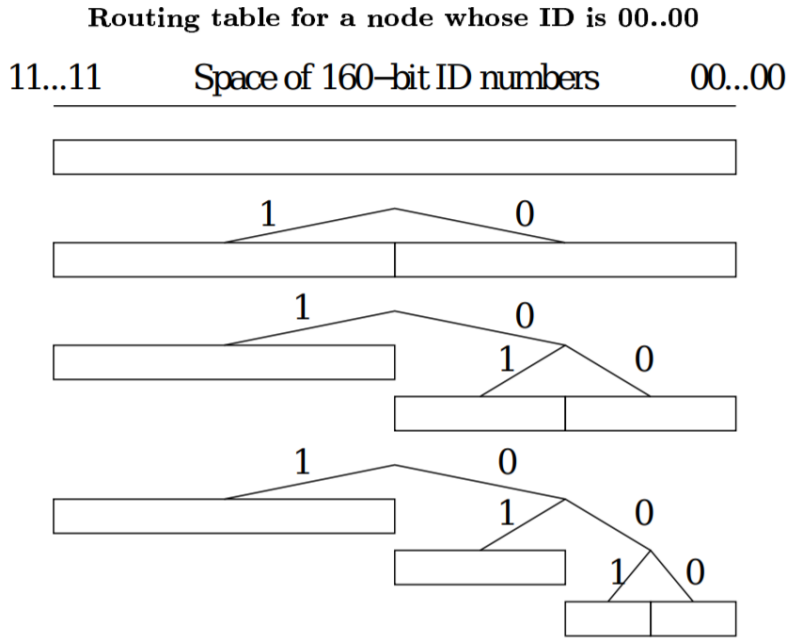
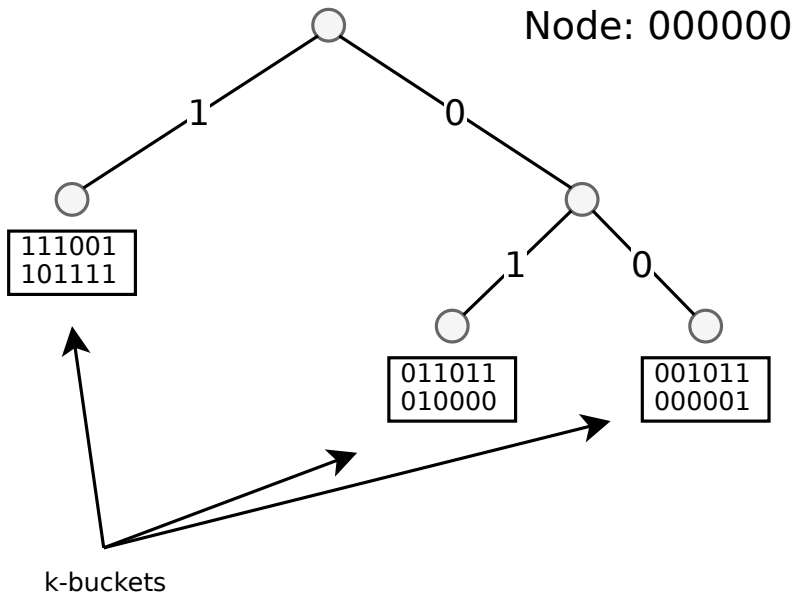


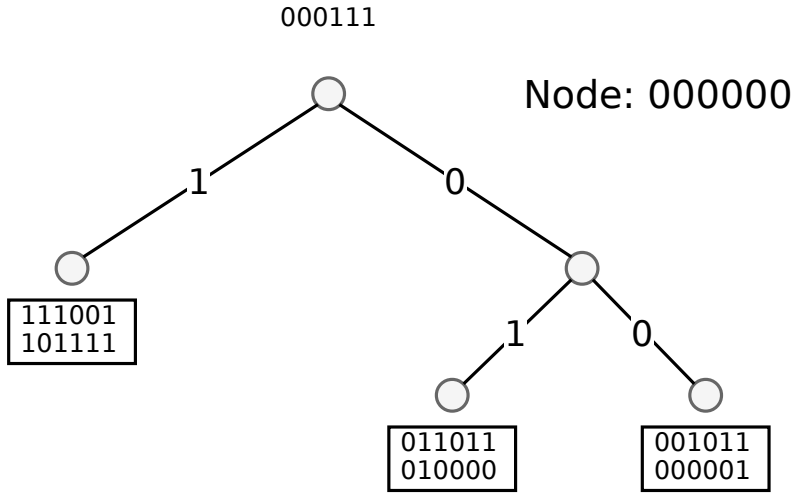
Fig. 4: Evolution of a routing table over time. Initially, a node has a single k-bucket, as shown in the top routing table. As the k-buckets fill, the bucket whose range covers the node's ID repeatedly splits into two k buckets.

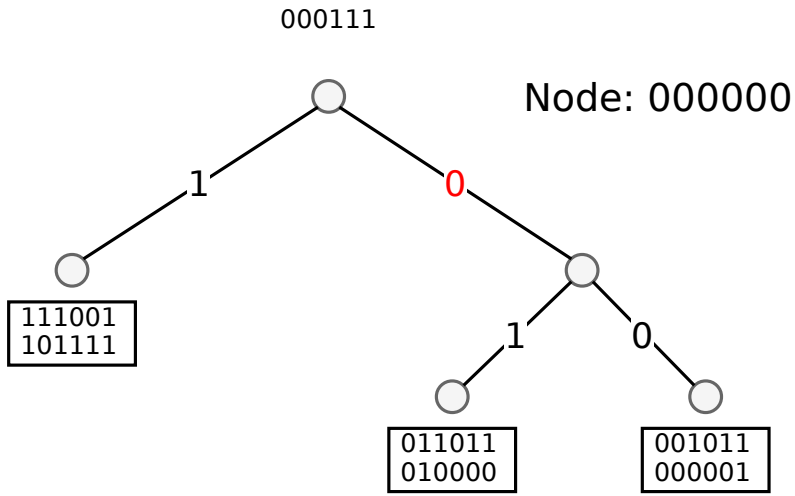
Example

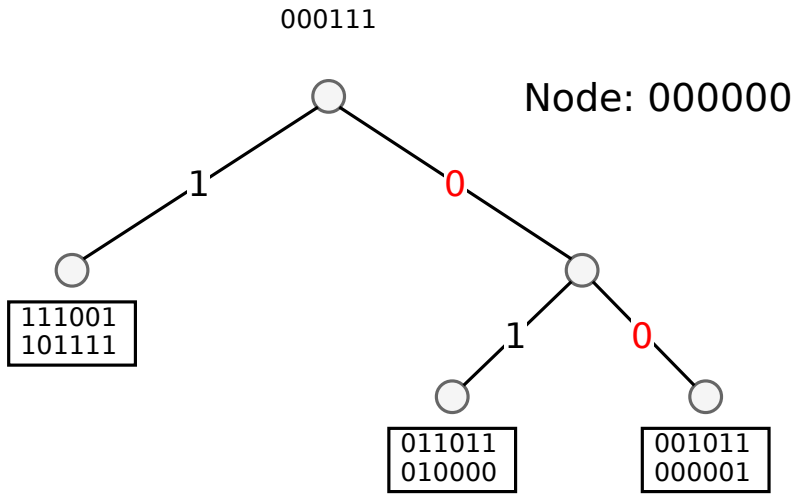
- k = 2
- $\alpha = 1$ (no asynchronous requests, also no asynchronous pings)
- Node identifier (000000) is not in the routing table

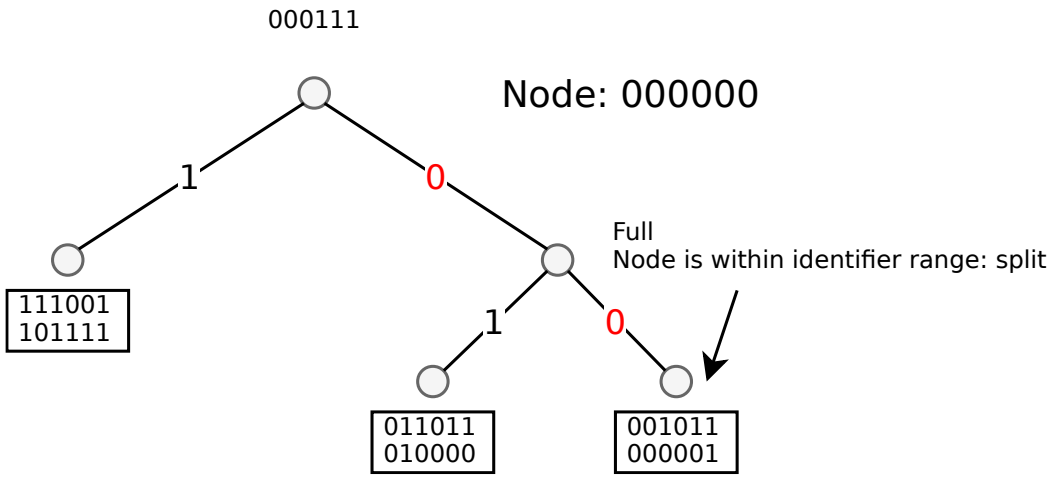


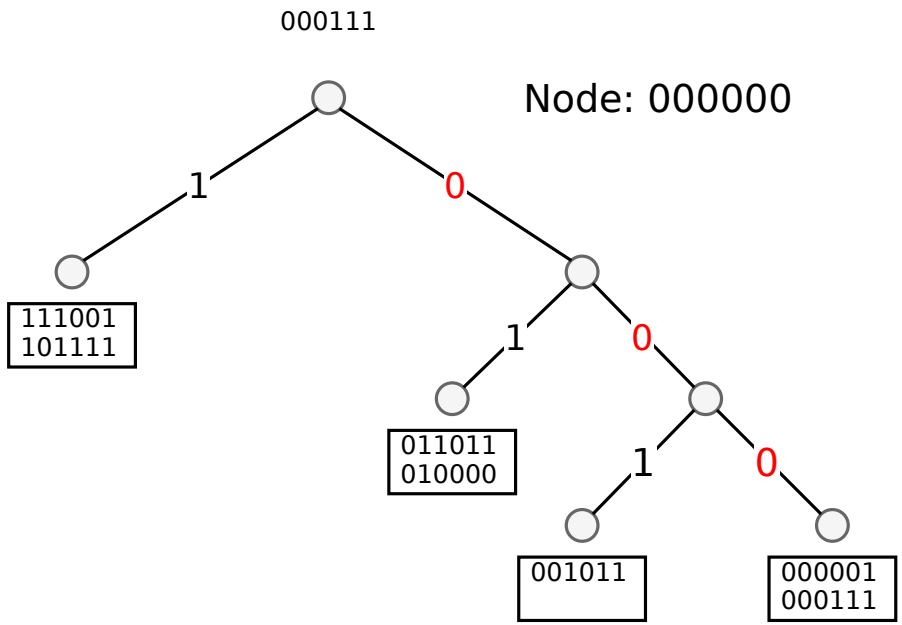
Node `000111` is involved with an RPC request, what happens?

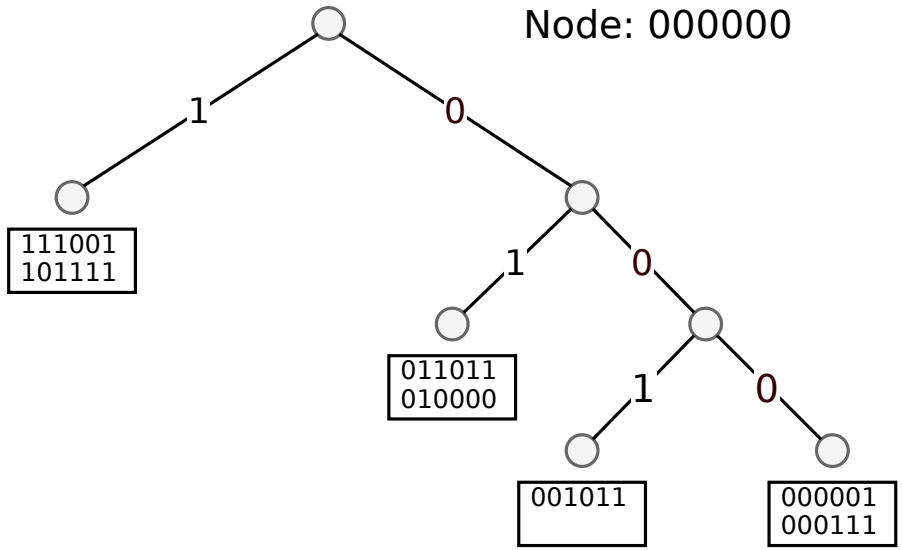



000111
Node: 000000
1
0
111001
101111
1
0
011011
010000
001011
000001

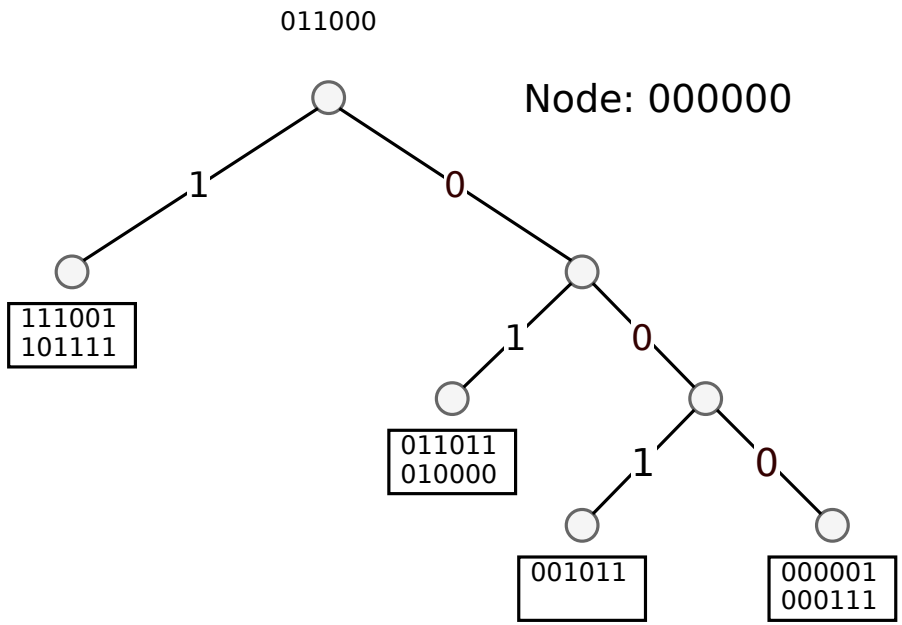

000111
Node: 000000
1
0
111001
101111
1
0
011011
010000
001011
000001

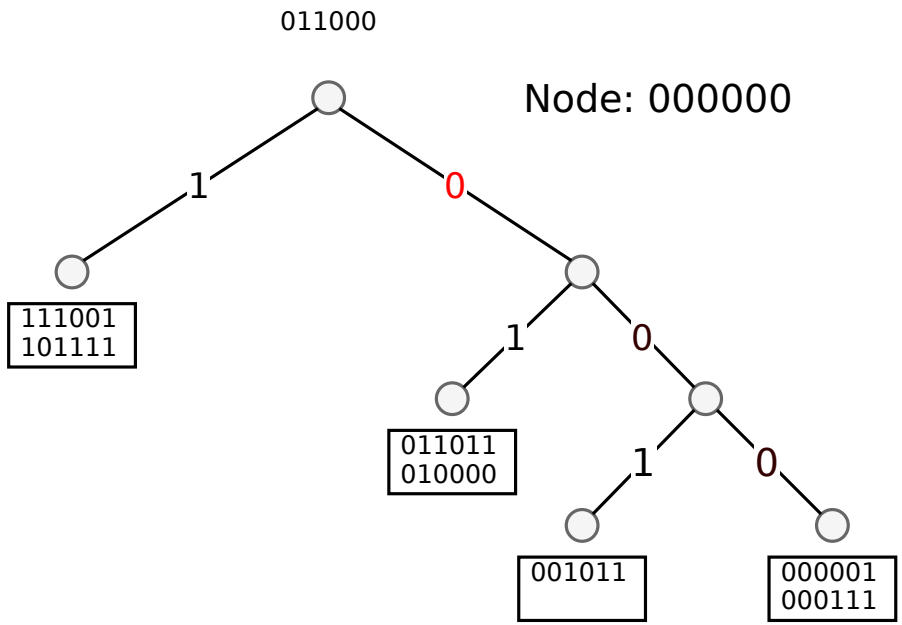

000111
Node: 000000
1
0
Full
Node is within identifier range: split
111001
101111
1
0
011011
010000
001011
000001

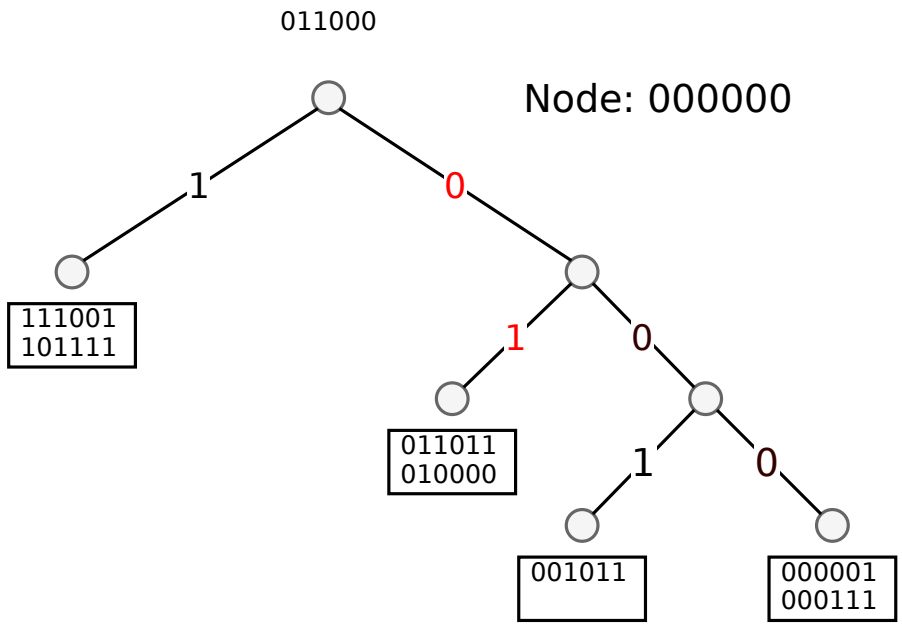

000111
Node: 000000
1
0
111001
101111
1
0
011011
010000
1
0
001011
000001
000111

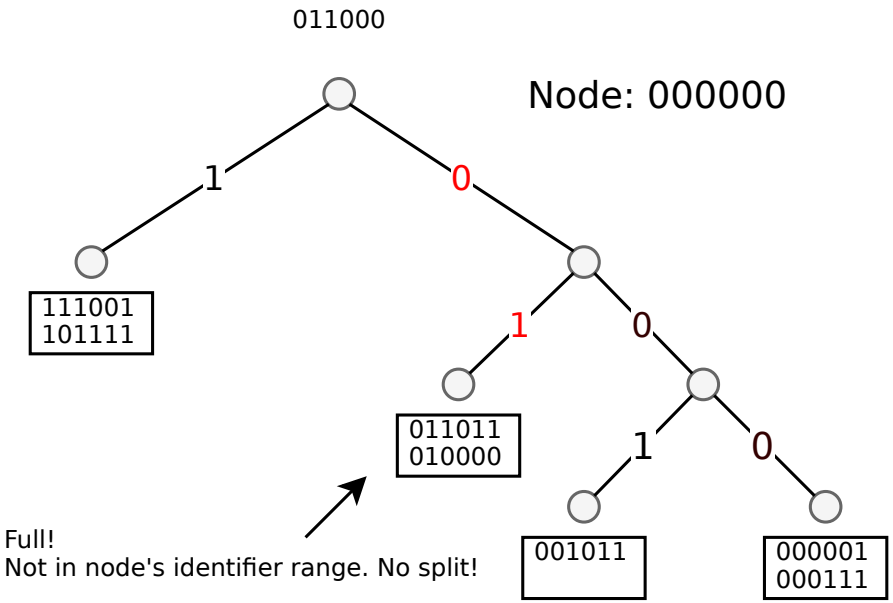

Node: 000000
1
0
111001
101111
1
0
011011
010000
1
0
001011
000001
000111

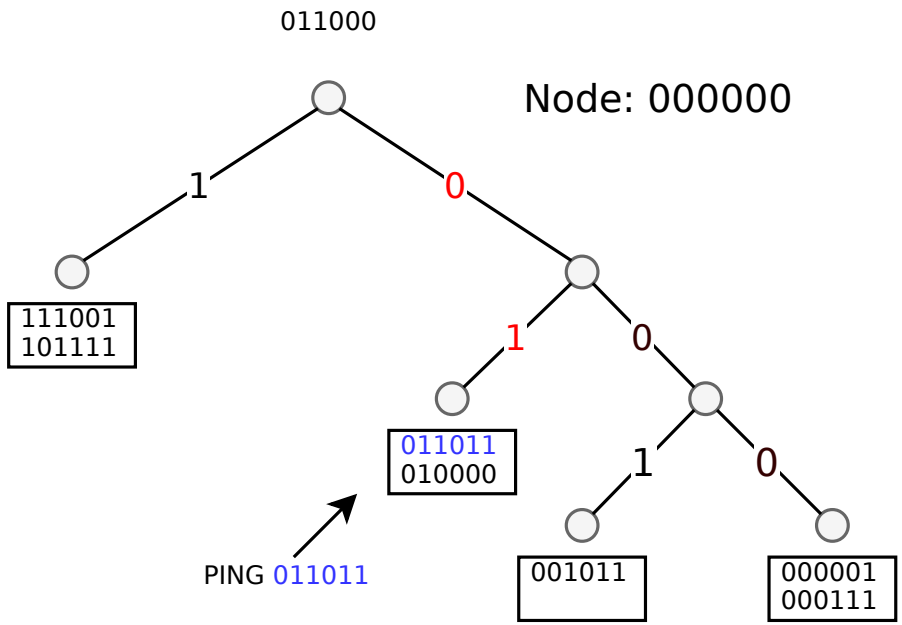
A new node `011000` is involved with a RPC message.

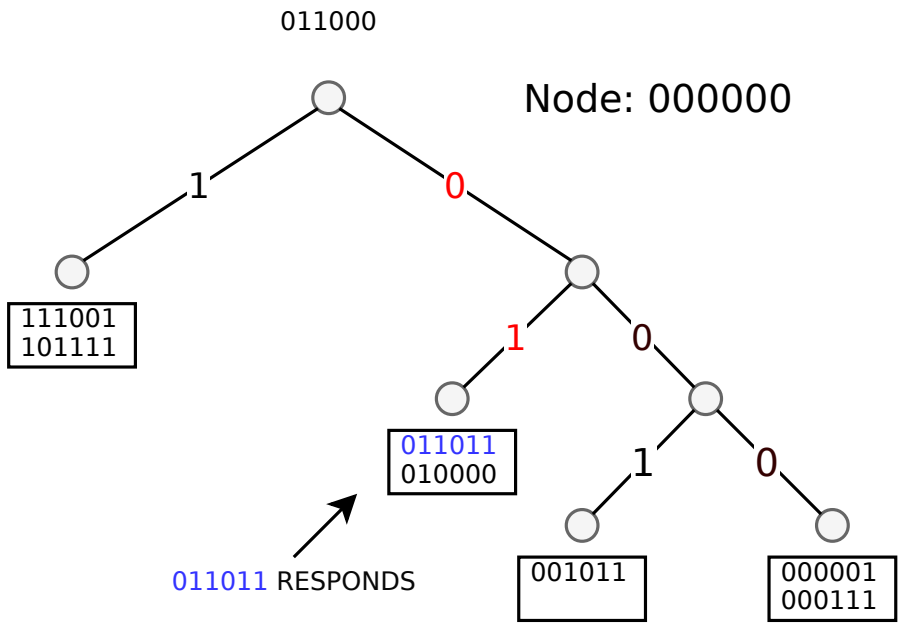


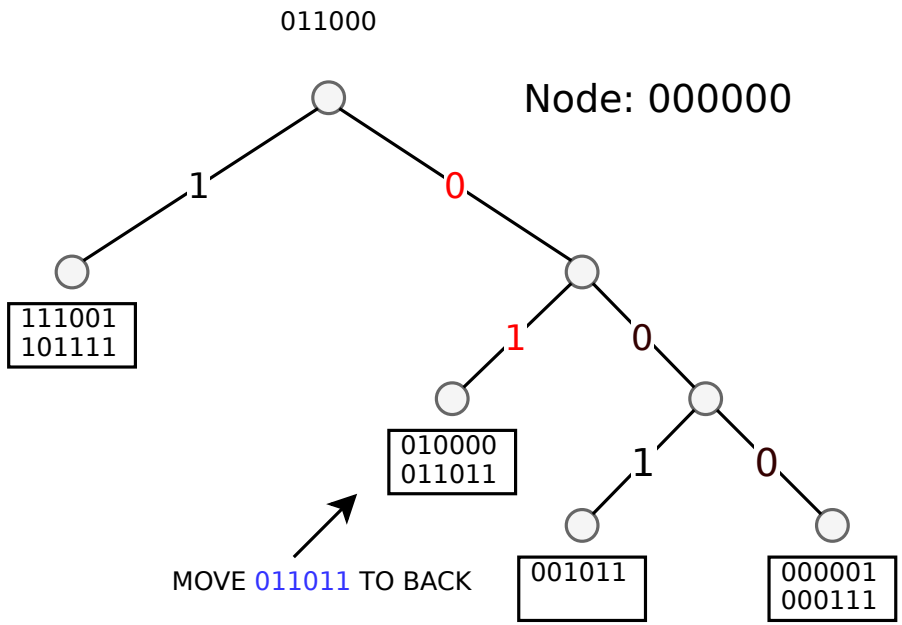

011000
Node: 000000
1
0
111001
101111
1
0
011011
010000
1
0
001011
000001
000111

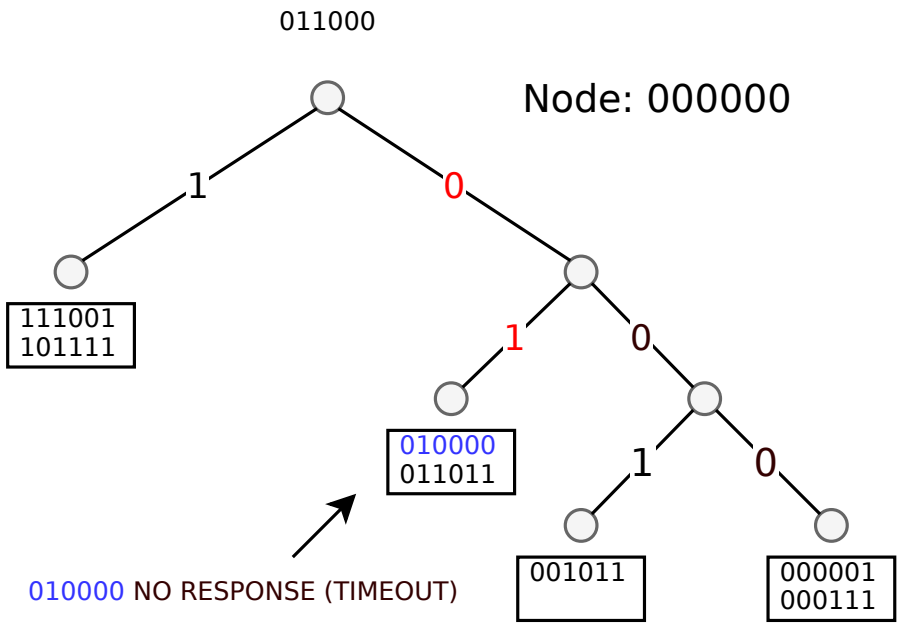

011000
Node: 000000
1
0
111001
101111
1
0
011011
010000
1
0
001011
000001
000111

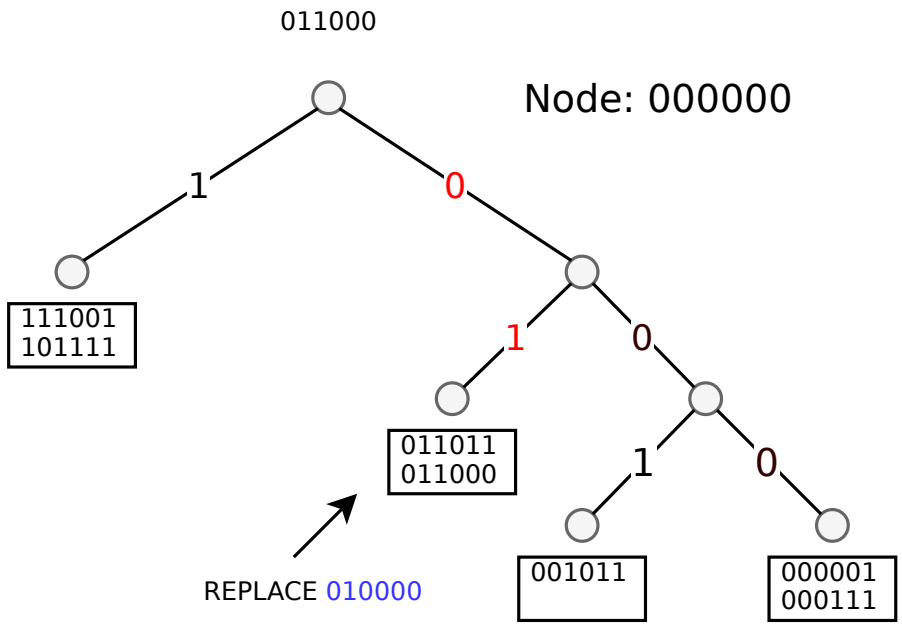

011000
Node: 000000
1
0
111001
101111
1
0
011011
010000
1
0
001011
000001
000111
Full!
Not in node's identifier range. No split!

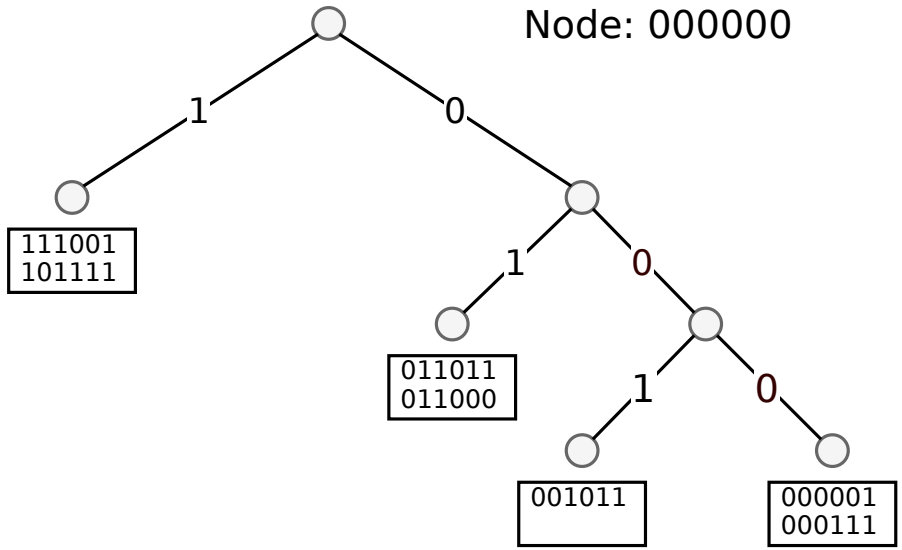

011000
Node: 000000
1
0
111001
101111
1
0
011011
010000
1
0
PING 011011
001011
000001
000111


011000
Node: 000000
1
0
111001
101111
1
0
011011
010000
1
0
001011
000001
000111
011011 RESPONDS


011000
Node: 000000
1
0
111001
101111
1
0
010000
011011
1
0
001011
000001
000111
MOVE 011011 TO BACK


011000
Node: 000000
1
0
111001
101111
1
0
010000
011011
1
0
001011
000001
000111
010000 NO RESPONSE (TIMEOUT)


011000
Node: 000000
1
0
111001
101111
1
0
011011
011000
1
0
001011
000001
000111
REPLACE 010000


Node: 000000
1
0
111001
101111
1
0
011011
011000
1
0
001011
000001
000111

Summary

- Efficient, guaranteed look-ups $\mathcal{O}(\log N)$
- XOR-based metric topology (provable consistency and performance).
- Possibly latency minimizing (by always picking the lowest latency note when selecting α nodes).
- Lookup is iterative, but concurrent (α).
- Kademlia protocol implicitly enables data persistence and recovery, no special failure mechanisms requires.
- Flexible routing table robust against DoS (route table flushing).

The end.

References

- Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review, 31(4), 149-160.
- Maymounkov, P., & Mazieres, D. (2002, March). Kademlia: A peer-to-peer information system based on the xor metric. In International Workshop on Peer-to-Peer Systems (pp. 53-65). Springer, Berlin, Heidelberg.