P2P Systems and Blockchains
Spring 2019,
instructor: Laura Ricci
laura.ricci@unipi.it

Lesson 5:
PREFIX-BASED DHT:
KADEMLIA

8/3/2019
PREFIX MATCHING DHT

- Plaxton, Rajamaran and Richa: Routing: prefix or plaxton routing:
  - a mechanism for the efficient diffusion of object over a network
  - published in 1997, before P2P systems came about!
- Basic ideas: a generalization of the routing on hypercubes.
  - map the nodes and the keys to numbers of m digits of a certain base
  - assign each key to the node with which it shares the longest prefix, if possible

- prefix matching DHT: a family of DHT
  - Pastry
  - Tapestry
  - Kademlia
IDENTIFIER SPACE

- $2^l$-bit identifiers (typically: $l = 128$) wrap-around at $2^l - 1 \rightarrow 0$
- Interpret identifiers to the base of $2^b$ (typically: $b = 4$, base 16)
- Identifiers described by a tree structure
  - Kademlia: $b=1$, binary tree
- pair keys on longer prefix matching nodes, if possible, or the node with numerically close Id.

![Diagram of Kademlia identifier space]

- $l=6$: 6-bit identifiers
- $b=2$: base 4
- key
- node
- managed by
The identifier tree is modelled by a tree

- depth of the tree \( \leq l \), \( l \) the length of the identifiers.
  - internal nodes correspond to identifier prefixes
- each node has \( b \) sons (one son for each digit in that base)
- leaves: keys & node Ids;
- the routing table of each node includes some references to nodes of this tree

\[ \text{l=6: 6-bit identifiers} \quad b=2: \text{base 4} \]
• $b =$ symbol size $=$ number of bits corrected at each step

• In Pastry $b=4$

• In Kademlia $b=1$, at each routing step at least a bit is corrected

• The size of the node routing table and the number of look-up hops depends on $b$
  • Routing: $O(\log_b(n))$

• Remark: some IP-level protocols may be considered prefix matching, where the matching is computed on the IP address of the nodes.
PLAXTON ROUTING TABLES

- **K-buckets**: list of references to nodes, stored in the routing table

- At each lookup step each node has the possibility to choose among K different contacts
  - K = 1 in Pastry
  - K ≈ 20 in Kademlia

- A value K > 1 guarantees
  - an higher robustness and tolerance to faults
  - possibility to choose among alternative routing paths
  - possibility to search the key in parallel on different paths
    - **parallel routing**: a key K received by a node is sent in parallel to a set of nodes, taken form the k-bucket

- Chord has a single contact (finger) in each row of the routing table: very strict constraint on the finger to insert
PLAXTON ROUTING STRATEGIES

- **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
P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric.

- but.... Where does the word “Kademlia” come from?
  What I know is—it is a Turkish word for a “lucky man” and, more importantly, is the name of a mountain peak in Bulgaria.
  [Petar Maymounkov, bulgarian guy]

- Protocol specification:

  http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html
THE KADEMLIA DHT


• “the de facto standard searching algorithm for P2P (peer-to-peer) networks on the Internet.”

• a protocol specification for decentralizing peer-to-peer network operations, efficiently storing and retrieving data across the network.
  • decentralized: data is not stored on a central server, but rather redundantly stored on peers.
  • fault tolerant: if one or more peers drops out of the network, the data, having been stored on multiple peers, should still be retrievable.
  • complicated database engines are not required: data stored is typically stored in key-value pairs, making it suitable for even IoT devices with limited storage to participate in the network.

its protocol is used by the largest public DHTs.

- KAD network (emule)
- BitTorrent Mainline DHT (MDHT)
- Ethereum

- a set of characteristics which are not offered by any existing DHT (Chord, Pastry,..)
  - routing information spreads automatically as a side-effect of lookups
  - flexibility to send multiple requests in parallel to speed up lookups by avoiding timeout delays (parallel routing)
  - iterative routing
“distance” between two objects: bitwise $\oplus$ (XOR) operation on their identifiers (160 bit space), interpreted as an unsigned integer

\[
\text{ID} = \text{sha-1 (Communication Breakdown)} = \text{a1174eb9d7b9150ac6077b3baa7d378486447a0d}
\]

**Node A**
\[
\text{ID}_A = \text{sha-1 (194.29.169.2)}: 67a83db6814412740c808c949761b9b4aec0a492
\]

**Node B**
\[
\text{ID}_B = \text{sha-1 (194.29.160.5)}: e6954b6744885214b25d3e41d5bc457bb3474
\]

**Node C**
\[
\text{ID}_C = \text{sha-1 (175.165.110.85)}: 3afbeba4271a07e962c87784a385538eba8a4882
\]

\[
\begin{align*}
\text{ID}_A \text{ XOR ID} &= \text{C6BF730F56FD077ECA87F7AF3D1C8E302884DE9F} \\
\text{ID}_B \text{ XOR ID} &= \text{478205DE9331471E7BD52CE84E606C40D1FF4E79} \\
\text{ID}_C \text{ XOR ID} &= \text{9BECA51DF0A312E3A4CF0CBF09F8640A3CCE328F}
\end{align*}
\]

Node B is the closest (4<C and 4<9) and should store \((\text{ID}, 94.29.160.5, 3465)\)
**XOR: IS REALLY A METRIC?**

- $d(x,y) > 0$
- $d(x,y) = 0$ if and only if $x = y$

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$p \oplus q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- $\forall x,y: d(x,y) = d(y,x)$ symmetry
- $d(x,y) \oplus d(y,z) = d(x,z)$ transitivity
- $d(x,y) + d(y,z) \geq d(x,z)$ triangular inequality
  - follows from the previous one
- given $x$ and a distance $\Delta$, it exists a single $y$ such that $d(x,y) = \Delta$ unidirectionality
  - $x = 1001$, $\Delta = 0001$, the only point at distance $D$ from $x$ is $y = 1000$
WHY THE XOR METRIC?

- any peer in the same half of the ID space as the starting peer is closer to it than any peer from the other half.
- distance graph looks the same in both halves but shifted along the y-axis.
- the same is true when further separating each half into smaller fractions.
WHY THE XOR METRIC?

- The metric is related to the identifier prefix:
  the larger is the prefix common to two nodes, the smaller is their distance computed by $\oplus$

- "close" nodes are characterized by a long common prefix

My Node ID: 11 => 1011
Bit-length = 4

- $d(11, 10)$
  \[
  \begin{array}{c}
  11: 1 0 1 1 \\
  \text{xor} 10: 1 0 1 0 \\
  \hline
  0 0 0 1 = 1
  \end{array}
  \]

- $d(11, 12)$
  \[
  \begin{array}{c}
  11: 1 0 1 1 \\
  \text{xor} 12: 1 1 0 0 \\
  \hline
  0 1 1 1 = 7
  \end{array}
  \]

- $d(11, 4)$
  \[
  \begin{array}{c}
  11: 1 0 1 1 \\
  \text{xor} 4: 0 1 0 0 \\
  \hline
  1 1 1 1 = 15
  \end{array}
  \]

More shared bit pre-fix = closer distance
Identifier tree and prefix matching:

- assign a key $K$ to the node whose identifier is closer to $K$, according to the $\oplus$ metrics
- key $\sqsubset$ is assigned to a leaf $\bullet$ in the left subtree, the closest leaf, while the numerically closest leaf is in the right subtree.
- according to $\oplus$ metric, the key is closer to any node in its subtree than to nodes in other subtrees
- look-up guided by the tree structure
WHY THE XOR METRIC?

• **symmetric:**
  - enables Kademlia to learn contacts from ordinary queries it receives.
  - helps in building the routing tables
  - non symmetric distances (Chord fingers are not symmetric) does not allow this

• **unidirectional:**
  - lookups for same key converge to the same path, and thus caching items along this path is good to avoid hotspots.
CHORD METRIC IS NOT SYMMETRIC

• Chord calculates the distance between two peers by subtracting the identifiers numerically from each other

• takes into consideration that the identifier space is “wrapped”

\[
distance(A, B) = B - A \quad \text{(for } B \geq A) \\
= B - A + 2^N \quad \text{(for } B < A) \\
\]

that is

\[
distance(A, B) = (B - A + 2^N) \mod 2^N
\]

• for instance:

\[
distance(0, 11) = (11 - 0 + 16) \mod 16 = (27) \mod 16 = 11 \\
distance(11, 0) = (0 - 11 + 16) \mod 16 = (5) \mod 16 = 5
\]
two leaves may be close in the tree and numerically close, but they are distant according to the metrics $\oplus$

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

numerical difference between 1000 and 0111 = 1
DISTANCES AND IDENTIFIER TREE

- consider two identifiers x and y of length L that
  - share a common prefix of length p
  - differ in the last \(i=L-p\) bits

- their distance (exclusive OR) will be such that
  \[2^{i-1} \leq d(x,y) < 2^i\]

\[
\begin{align*}
X &= 0 \ 1 \ 0 \ 1 \ 1 \ 0 \\
Y &= 0 \ 1 \ 1 \ 1 \ 1 \ 0 \\
X \oplus Y &= 0 \ 0 \ 1 \ 0 \ 0 \ 0, \quad d(x,y) = 2^3=8 \text{ (minimal distance).}
\end{align*}
\]

\[
\begin{align*}
X &= 0 \ 1 \ 0 \ 1 \ 1 \ 0 \\
Y &= 0 \ 1 \ 1 \ 0 \ 0 \ 1 \\
X \oplus Y &= 0 \ 0 \ 1 \ 1 \ 1 \ 1, \quad d(x,y) = 2^4-1 = 15 \text{ (maximal distance)}
\end{align*}
\]

- this enables to pair the nodes of the subtree with an identifier range
DISTANCES AND IDENTIFIER TREE

- consider a leaf in the left subtree and a leaf in the right subtree
- the length of the shared prefix is $= 0$
- the distance varies:

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

$$\begin{align*}
0111 & \oplus 1000 = 1111 = 15 \text{ (maximal, numerical distance is minimal)} \\
0111 & \oplus 1111 = 1000 = 8 \text{ (minimal, numerical difference is high)}
\end{align*}$$
The distance between the two nodes in the figure is minimal: they differ in only the last bit:

\[ 2^0 \leq d < 2^1 \]

\[ 0110 \oplus 0111 = 0001 = 1 \]
- peers present in the network are much lesser than the identifiers: the space of identifier is very large and not all the identifiers are paired with a peer.
- node tree: an unbalanced binary tree showing only the identifiers of peers present in the network (a subset of all the identifiers).
- a leaf for each peer: the identifier of the leaf is a prefix of that of the peer.
A leaf in the node tree may correspond to a prefix $P$ of the identifier of a peer if the overlay includes a single peer with that prefix $P$.

- $0011$ uniquely identifies the red peer
- no other peer in the overlay with the same prefix
- the rest of the path is not useful to identify the peer
Routing table, the basic idea:

- maintains some contacts for each subtree/common prefix
- the figure, shows the contacts in the routing table of the red peer
- It includes 2 contacts for each subtree
THE ROUTING TABLE

• routing table: the rows are **k-buckets**, each one contains k contacts

• for any $0 \leq i < 160$, the i-th row of the routing table of node ID contains up to k-buckets such as:

$d_{\text{XOR}}(\text{node ID}, \text{contact ID}) \in [2^i, 2^{i+1})$

• each k-bucket corresponds to a subtree

• each row stores:

  (ID, IP, UDP port)
ROUTING TABLE

- 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
  - may include a few contacts

- the last entries of the routing table correspond to peers sharing a smaller prefix, and cover a larger set of identifiers
  - may include a larger number of contacts, 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

- nodes in each bucket are maintained ordered such that:
  - least recently contacted nodes are in the first positions of the list
K-BUCKETS MANAGEMENT: ADD CONTACT

1. Find k-bucket for sender’s node ID.
2. Does the node exist?
   - Yes: Promote sender’s node to tail of list.
   - No: Go to next step.
3. Bucket Full?
   - No: Discard sender.
   - Yes: Ping least recently seen node.
4. Response?
   - No: Evict least recently seen node.
   - Yes: Promote least recently seen node to tail of list.

Done.
ADD CONTACT: PSEUDO CODE

• K-buckets: lists, least-recently seen nodes in the first positions.
• when a node receives any message from another node, it updates the appropriate k-bucket for the sender’s node ID

if the sending node already exists in the k-bucket:
    move it to the tail of the list.
otherwise
    if the bucket has fewer than k entries:
        insert the new sender at the tail of the list
    otherwise
        ping the k-bucket’s least-recently seen node
        if the least-recently seen node fails to respond
            evict it from the k-bucket and insert
            the new sender at the tail.
        otherwise move it to the tail of the list,
            and the new sender’s contact is discarded.
Esempio:

Tabella nodo 0

<table>
<thead>
<tr>
<th>m</th>
<th>k</th>
<th>Nodo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>nod01</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>nod02</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>nod03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nod04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nod05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nod06</td>
</tr>
</tbody>
</table>

m=3; k=2

1. 6 manda un messaggio a 0, ma il k-bucket è pieno (k=2)
2. 0 interroga 4
3. 4 risponde e viene spostato al fondo (6 è fuori)
Esempio:

Tabella nodo 0

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[1,2)</td>
</tr>
<tr>
<td>1</td>
<td>[2,4)</td>
</tr>
<tr>
<td>2</td>
<td>[4,0)</td>
</tr>
</tbody>
</table>

m=3; k=2

1. 6 manda un messaggio a 0, ma il k-bucket è pieno (k=2)
1. 0 interroga 4 (nodo in testa)
1. 4 non risponde: 6 è messo in fondo al k-bucket
K-BUCKETS MANAGEMENT: MOTIVATIONS

Preference for old contacts with respect to newer ones, motivations:

- driven by an analysis of collected Gnutella trace data collected
  - the longer a node has been up, the more likely it is to remain up another hour.
  - by keeping the oldest live contacts around, k-buckets maximize the probability that the nodes they contain will remain online.

- second benefit of k-buckets
  - resistance to certain DoS attacks.
  - an attacker cannot flush the nodes’ routing state by flooding the system with new nodes.
  - Kademlia nodes will only insert the new nodes in the k-buckets when old nodes leave the system.”
Why this politics for bucket management?

- x axis: minutes
- y axis: percentage of peers that, being on-line for x minutes, are on line also for next x+60 minutes

percentage of nodes in a Gnutella network which remains online in the next hour as a function of their previous uptime.

- the longer a node remains on-line, the higher is the probability that it remains online for a even longer interval of time
• Why this politics for bucket management?
  • always prefer the contacts which are have been present in the network for a longer time, because, with high probability, they will remain in the network also in the next period.
  • least recently seen eviction
    • deleted nodes are those in the first position of the k-buckets list, that are those least recently contacted.
K-BUCKETS PERIODIC REFRESHMENT

- 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.
- However, it may be the case that a k-bucket is not refreshed for a given period of time, due to the lack of messages from nodes in the range covered by the k-bucket.
- For this reason, a refresh is periodically executed (once each hour):
  - Kademlia chooses an identifier belonging to the range covered by the bucket at random and searches that identifier.
  - If the node with that identifier sends a reply, it is inserted in the k-bucket.
Kademlia protocol consists of 4 primitive (not iterative) operations, defined as Remote Procedure Calls (RPCs) which exploits UDP:

- **PING v→w**
  - probe node w to see if its online

- **STORE v→w (Key, Value)**
  - instructs node w to store a <key, value> pair

- **FIND_NODE v→w (T)**
  - the recipient of the message (w) returns (IP address, UDP port, Node ID) triples for the k nodes it knows about 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.
  - in any case, the recipient must return k items, unless there are fewer than k nodes in all its k-buckets combined, in which case it returns every node it knows about.
Kademlia protocol consists of 4 primitive (not iterative) operations, defined as Remote Procedure Calls (RPCs) which UDP:

- **FIND_VALUE** \( v \rightarrow w(T) \)
  - In: \( T \), 160-bit ID
  - Out:
    - if a value corresponding to \( T \) is present in the queried node \( w \), the associated data is returned
    - otherwise it is equivalent to **FIND_NODE** and \( w \) returns a set of \( k \) triples
  - If **FIND_VALUE** returns a list of other peers, it is up to the requester to continue searching for the desired value from that list
KADEMLIA: NODE LOOKUP

- procedure to locate the $k$ closest nodes to some given node ID.
  - Kademlia employs a recursive algorithm for node lookups, based on the basic protocol operation `FIND_NODE`

- lookup is exploited for
  - finding nodes
  - finding values. If it’s finding values, it needs to stop if/when the value is found.

- $\alpha$ is a system-wide concurrency parameter
  - it controls how many `FIND_NODE` or `FIND_VALUE` are executed in parallel
  - when $\alpha = 1$, the lookup algorithm is similar to Chord in terms of message cost and the latency of detecting failed nodes.
  - however, Kademlia can route for lower latency because it has the flexibility of choosing any one of $k$ nodes to forward a request to.
Chord routing table is rigid, has only one way information flow
- complicates recovery process
- incoming traffic cannot be used for reinforcing routing table.
- less fault-tolerance
NODE LOOK UP

\( k\text{-closest} = \alpha \) contacts from the non-empty \( k\)-bucket closest to the key

\textbf{if} there are fewer than \( \alpha \) contacts in that bucket \textbf{then}

\( k\text{-closest} = k\text{-closest} \cup \) closest contacts from other buckets.

\( closestNode = \) the closest node in \( k\text{-closest} \)

\(/\ast\) recursive step

\texttt{repeat}

\texttt{select} from \( k\text{-closest} \), \( \alpha \) closest contacts which have not been queried yet

\texttt{send} in parallel, asynchronously \texttt{FIND\_NODE} to the contacts in \( k\text{-closest} \)

\texttt{each contact, if live, returns} \( k \) nodes

\texttt{add} to \( k\text{-closest} \) the new received nodes and update \( closestNode \)

\texttt{until} no node closer to the target than \( closestNode \) is returned

\texttt{send} in parallel asynchronously \texttt{FIND\_NODE} to the \( k \) closest nodes it has not already queried

\texttt{return} the \( k \) closest nodes
The algorithm that Kademlia uses for locating the k nodes nearest to a key:

- is iterative
- exploits the basic function defined in the previous slides
- intuitively, at each iteration, the XOR metric is reduced by $\frac{1}{2}$ and results in smaller size k-buckets
LOOK UP PROCEDURE AT A GLANCE

- \( \alpha \): number of nodes to which the query is propagated, at each routing step
- iterative Routing
- an example of routing for \( \alpha = 1 \)
LOOK UP PROCEDURE AT A GLANCE

Black node : query source (0011)
Orange Node : query target(1110)
Green Node : nodes known from a bucket of the black nodes

Prefix Match Routing: at each step the common prefix with the key increases
LOOK UP PROCEDURE AT A GLANCE

- **Black node**: query source (0011)
- **Orange Node**: query target (1110)
- **Green Node**: nodes known form the back nodes (in its routing table)
- **Blue Node**: returns further nodes to the black node

Prefix Match Routing: at each step the common prefix with the key increases
LOOK UP PROCEDURE AT A GLANCE

Black node : query source (0011)
Orange Node : query target(1110)
Green Node : nodes known form the back nodes (in its routing table)
Blue Node : returns further nodes to the black node

Prefix Match Routing: at each step the common prefix with the key increases
LOOK UP PROCEDURE AT A GLANCE

Black node : query source (0011)
Orange Node : query target (1110)
Green Node : nodes known form the back nodes (in its routing table)
Blue Node : returns further nodes to the black node

Prefix Match Routing: at each step the common prefix with the key increases
LOOK UP PROCEDURE AT A GLANCE

The blue node 0011 looks for the red node 1101.
It has a reference to the green nodes 1001 e 1110, and
\[ \text{dist}(1101, 1001) = 4, \text{dist}(1101, 1110) = 3 \]
it is 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 the target

parallel routing: the blue node sends the request to both nodes
LOOK UP: AN OBSERVATION

• Consider a node $x$ and two further nodes $y$ and $z$, with
  \[ \text{dist}(y, x) < \text{dist}(z, x) \]
  $z$ knows $x$, while $y$ does not know $x$.

• The dispatch of the query to the node which is closest to the target not
  necessarily implies the smaller path toward the target.

• Routing: the query is sent to the $\alpha > 1$ nodes closest to the target.

• The unidirectionality of the $x$-or metric guarantees that all the paths converge
  toward the target.
P looks for the key $Q$ (which is the identifier of a node or of a content)

- looks in the bucket-list the $a$ nodes closest to $Q$
- looks in the $k$-bucket closest to the key and not empty. If it includes less than $a$ nodes ($a=3$), looks in close buckets
- selected contacts may belong to different $k$-buckets
Here are stored the nodes closest to Q

P selects $\alpha$ nodes from the selected bucket
P sends the query in parallel to all the selected nodes, through the 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.
Iterative Routing:

- each node returns the results to P
- the results are inserted in a list which is ordered on the basis of the distance between the node and Q
- P continues the routing process through the results obtained from P
LOOK UP PROCEDURE: A COMPLETE EXAMPLE

P updates its k-buckets list with the received information

Information received A, B, C

P again selects $\alpha$ nodes from the received information

If it obtains nodes closer to the target with respect to the preceding nodes, it performs look-up on these nodes

Otherwise, it chooses further nodes from those which have not been contacted before
Terminate when a round of FIND_NODE(T) fails to return any closer nodes
NODE STORE

• to store a (key,value) pair
  • a participant perform a look-up to find the k closest nodes to the key and sends them STORE RPCs.
  • then, data is replicated on these nodes

• in any case, consider following scenarios
  • some of the k nodes (or all) that initially get the (key,value) pair leave the network
  • new nodes enters the network with an identifier closer to some published key than the nodes on which the key-value pair was originally published

• re-publishing mechanism:
  • each node re-publishes (key,value) pairs as necessary to keep them alive.

• For Kademlia’s file sharing application, the original publisher of a (key,value) pair is required to republish it every 24 hours. Otherwise, (key,value) pairs expire 24 hours after publication.
NODE JOIN

- new (joining node) borrow an alive node’s ID off-line (bootstrap node boot)
- initial routing table of new has a single k-bucket containing new and boot.
- new performs FIND_NODE(new) through boot to learn about other nodes
  - finds some nodes close to itself. Some low index k-buckets are filled
- other nodes start to know new and insert it in their routing tables
- new performs FIND_NODE(ID) for identifier ID in k-buckets further away than its own k-bucket generate a node identifier at random
- k-buckets are subsequently enriched with the information received in the queries passing through new

Note the flexibility of this procedure with respect to the joining procedure of Chord!
KADEMLIA PROTOCOL

Maintenance
• refresh k-buckets for which there was no contact within a certain time, e.g. an hour
• refresh means lookup of random ID in bucket.

Storage & Caching
• to store a value, locate the k closest nodes to the ID of the node via lookup and then store (replicate) the value at these nodes.
• values are considered soft-state and need refreshing.
• values are cached at the first node on a path that did not know it.

Leave
• the node leave does not require further operations
  • if a node does not reply, it will be discarded from the k-buckets
PERIODIC TASKS

- Each node periodically publishes the pair <key,values> to guarantee the persistence of the data inserted in the overlay.

- The periodic publication mechanism has been introduced for:
  - avoid data loss as a consequence of the voluntary leave or of the crash of a node.

- Some optimizations are defined to decrease the number of exchanged messages:
  - if a node receives a STORE, it suppose that the STORE has been sent to the close neighbours and does not publish again the key in the next hour.
Kademlia defines a flexible routing table:
- symmetric distances
- alternative paths toward a node, possibility of parallel lookups
- managing the routing table has a lower cost
- locality: store round-trip-time together each contact and choose the contact with lower round trip time

The symmetric metrics enables each node to enrich its routing table through the query.

On the contrary, in Chord:
- if a node x receives a query from y, y has in its finger table a reference to x, but x may be not a finger of y.
- the information included in a received query cannot, in general, be exploited to enrich the finger table.
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)
<table>
<thead>
<tr>
<th>CAN</th>
<th>Chord</th>
<th>Kademlia</th>
<th>Koorde</th>
<th>Pastry</th>
<th>Tapestry</th>
<th>Viceroy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Foundation</strong></td>
<td>Multi-dimensional space (d-dimensional torus)</td>
<td>Circular space (hypercube)</td>
<td>de Bruijn graph</td>
<td>Plaxton-style mesh (hypercube)</td>
<td>Plaxton-style mesh (hypercube)</td>
<td>Butterfly network</td>
</tr>
<tr>
<td><strong>Routing function</strong></td>
<td>Maps (key,value) pairs to coordinate space</td>
<td>Matching key and nodeID</td>
<td>Matching key and nodeID</td>
<td>Matching key and prefix in nodeID</td>
<td>Suffix matching</td>
<td>Routing using levels of tree, vicinity search</td>
</tr>
<tr>
<td><strong>System parameters</strong></td>
<td>Number of peers N, number of dimensions d</td>
<td>Number of peers N</td>
<td>Number of peers N, base of peer identifier B</td>
<td>Number of peers N</td>
<td>Number of peers N, base of peer identifier B</td>
<td>Number of peers N</td>
</tr>
<tr>
<td><strong>Routing performance</strong></td>
<td>$O(dN^d)$</td>
<td>$O(\log N)$</td>
<td>$O(\log \log N) + \text{small constant}$</td>
<td>Between $O(\log \log N)$ and $O(\log N)$, depending on state</td>
<td>$O(\log \log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td><strong>Routing state</strong></td>
<td>$2d$</td>
<td>$\log N$</td>
<td>$\log \log N + B$</td>
<td>From constant to $\log N$</td>
<td>$2\log \log N$</td>
<td>$\log \log N$</td>
</tr>
<tr>
<td><strong>Joins/leaves</strong></td>
<td>$2d$</td>
<td>$(\log N)^2$</td>
<td>$\log \log N + \text{small constant}$</td>
<td>$\log N$</td>
<td>$\log \log N$</td>
<td>$\log \log N$</td>
</tr>
</tbody>
</table>

DHT: COMPARISON

Laura Ricci
Prefix Based DHT: Kademlia