Lesson 8

PREFIX-BASED DHT: KADEMLIA

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

Laura Ricci
17/03/2017
PREFIX MATCHING DHT

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

\[
b = 4 \text{ and } m = 6:\]

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

Also called prefix-based routing or Plaxton’s Mesh.
**IDENTIFIER SPACE**

- $2^L$-bit identifiers (typically: $L = 128$) wrap-around at $2^L - 1 \leftrightarrow 0$
- interpret identifiers to the base of $2^b$ (typically: $b = 4$, base 16)
  - Kademlia: $b=2$, binary tree
- prefix-based tree topology
- Leaves: keys & node IDs;
  - pair keys on longer prefix matching nodes, if possible, or the node with numerically close Id.

![Diagram of Kademlia prefix-based tree topology]

- $l=6$: 6-bit identifiers
- $b=2$: base 4
- key
- node
- managed by
IDENTIFIER SPACE

The identifier tree is modeled by a tree

- depth of the tree $\leq l$, $l$ the length of the identifiers.
  - internal nodes: identifier prefixes
- each node has $b$ sons
- the routing table of each node includes some references to nodes of this tree
**ROUTING TABLE**

- Basic idea: the routing table of a node with ID maintains at least one link to nodes sharing a prefix a length $l$ with ID and differing in the $l+1$ digit.

- In the figure an example with $b=2$ (binary tree):
  - nodeID = 011010
  - links to 1*, 00*, 010*, 0111*, 01100*, 011011
THE ROUTING TABLES

- Linearise the tree to define the routing table of node N with identifier ID

- RoutingTable[i,j] contains:
  - if $j \neq (i+1)$th digit of ID
    - the reference to the identifier of a node K which
      - shares a prefix of length $i$ with N,
      - differs in digit $i+1$
      - is characterized by having digit $i+1 = j$
  - otherwise is empty: look at the next row!

for $b = 4$, $m = 6$, nodeID = 110223; routing table:

<table>
<thead>
<tr>
<th>$p = 0$</th>
<th>$d = 0$</th>
<th>$d = 1$</th>
<th>$d = 2$</th>
<th>$d = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = 0$</td>
<td>032130</td>
<td>1</td>
<td>210231</td>
<td>303213</td>
</tr>
<tr>
<td>$p = 1$</td>
<td>103002</td>
<td>1</td>
<td>123011</td>
<td>133233</td>
</tr>
<tr>
<td>$p = 2$</td>
<td>0</td>
<td>111210</td>
<td>112301</td>
<td>113331</td>
</tr>
<tr>
<td>$p = 3$</td>
<td>110031</td>
<td>110122</td>
<td>2</td>
<td>110310</td>
</tr>
<tr>
<td>$p = 4$</td>
<td>110200</td>
<td>110212</td>
<td>2</td>
<td>110232</td>
</tr>
<tr>
<td>$p = 5$</td>
<td>110220</td>
<td>110221</td>
<td>110222</td>
<td>3</td>
</tr>
</tbody>
</table>

$p =$ prefix length

$d =$ i+1 digit
THE ROUTING TABLES

Tab[i,j]
- may include a single pointer, like in Pastry, or a set of pointers (k-buckets), like in Kademlia
- more choices for each entry: for instance, it is possible to select the node that is geographically closer to the current node.

\[
\text{for } b = 4, \ m = 6, \ \text{nodeID} = 110223; \ \text{routing table:}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
p = 0 & d = 0 & d = 1 & d = 2 & d = 3 \\
\hline
\hline
\text{p = 0} & 032130 & 1 & 210231 & 303213 \\
\hline
\text{p = 1} & 103002 & 1 & 123011 & 133233 \\
\hline
\text{p = 2} & 0 & 111210 & 112301 & 113331 \\
\hline
\text{p = 3} & 110031 & 110122 & 2 & 110310 \\
\hline
\text{p = 4} & 110200 & 110212 & 2 & 110232 \\
\hline
\text{p = 5} & 110220 & 110221 & 110222 & 3 \\
\hline
\end{array}
\]

\text{p = prefix length}
\text{d = i+1 digit}
Critical property

- for larger row numbers the number of possible choices decreases exponentially

- in row i+1 we have $1/b$ the choices we had in row i, because we have more contraints on the prefix

- for larger row numbers the distance to the nearest neighbor increases exponentially

- the distance of the source to the target is approximately equal to the distance in the last step. As a result it is well approximated
Move closer to the target one digit at the time

<table>
<thead>
<tr>
<th>p</th>
<th>d = 0</th>
<th>d = 1</th>
<th>d = 2</th>
<th>d = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>032130</td>
<td>1</td>
<td>210231</td>
<td>303213</td>
</tr>
<tr>
<td>1</td>
<td>103002</td>
<td>1</td>
<td>123011</td>
<td>133233</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>111210</td>
<td>112301</td>
<td>113331</td>
</tr>
<tr>
<td>3</td>
<td>110031</td>
<td>110122</td>
<td>2</td>
<td>110310</td>
</tr>
<tr>
<td>4</td>
<td>110200</td>
<td>110212</td>
<td>2</td>
<td>110232</td>
</tr>
<tr>
<td>5</td>
<td>110220</td>
<td>110221</td>
<td>110222</td>
<td>3</td>
</tr>
</tbody>
</table>
Move closer to the target one digit at the time

<table>
<thead>
<tr>
<th>d = 0</th>
<th>d = 1</th>
<th>d = 2</th>
<th>d = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = 0</td>
<td>032130</td>
<td>1</td>
<td>210231</td>
</tr>
<tr>
<td>p = 1</td>
<td>103002</td>
<td>1</td>
<td>123011</td>
</tr>
<tr>
<td>p = 2</td>
<td>0</td>
<td>111210</td>
<td>112301</td>
</tr>
<tr>
<td>p = 3</td>
<td>110031</td>
<td>110122</td>
<td>2</td>
</tr>
<tr>
<td>p = 4</td>
<td>110200</td>
<td>110212</td>
<td>2</td>
</tr>
<tr>
<td>p = 5</td>
<td>110220</td>
<td>110221</td>
<td>110222</td>
</tr>
</tbody>
</table>
Prefix Matching Routing Lookup: N receive a look-up for key K. N searches in its routing table for:

- a node M whose ID has a b-base configuration matching the longest prefix with K
- sends K to M

- routing
  - at each step increase the length of the 'match' between the identifier of the key and that of the node.
  
- stops as soon as it is not possible to detect in the routing tables a longer match
  - the node closest to the key has been reached.
Move closer to the target one digit at the time

locate 322210

110223  303213  322001
Move closer to the target one digit at the time

locate
322210

110223  303213  322001  322200
Move closer to the target one digit at the time

locate 322210

110223  303213  322001  322200  322213
• $b = \text{symbol size} = \text{number of bits corrected at each step}$

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

• Pastry exploits $b=4$

• The size of the node routing table and the number of look-up hops depends on $b$

• Routing: $O(\log_b (n))$

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

- **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
  - $K \approx 20$ 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 MESH: 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
THE KADEMLIA DHT

• initially proposed by P. Maymounkov e D. Mazières (University of New York)

• its protocol is used by the largest public DHTs
  • KAD network (emule)
  • Vuze network,
  • BitTorrent Mainline DHT (MDHT),
    BEP 5 specification

• 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
THE XOR METRIC

“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**
\[ \text{ID}_B = \text{sha-1 (194.29.160.5): e6954b6744885214b2d257d3e41d5bc457bb3474} \]

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

**Node A XOR ID** = C6BF730F56FD077ECA87F7AF3D1C8E302884DE9F
**Node B XOR ID** = 478205DE9331471E7BD52CE84E606C40D1FF4E79
**Node C XOR ID** = 9BECA51DF0A312E3A4CF0CBF09F8640A3CCE328F

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

- $d(x,y) > 0$

<table>
<thead>
<tr>
<th></th>
<th></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>

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

- $\forall \: x, y: \: d(x,y) = d(y,x)$ symmetry

- $d(x,y) \oplus d(y,z) = d(x,z)$ transitivity

- $d(x,y) + d(y,z) \geq d(x,z)$ triangular inequality
  - the triangle inequality follows from the following facts:
    - $d(A, C) = d(A, B) \oplus d(B, C)$
    - $\forall \: A \geq 0, \: B \geq 0 \: A + B \geq A \oplus B$

- 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 $\Delta$ 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.
CHORD METRIC IS NOT SYMMETRIC

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

- Take into consideration that the identifier space is “wrapped”

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

That is

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

- For instance:
  \[
  \begin{align*}
  \text{distance}(0, 11) &= (11 - 0 + 16) \mod 16 = (27) \mod 16 = 11 \\
  \text{distance}(11, 0) &= (0 - 11 + 16) \mod 16 = (5) \mod 16 = 5
  \end{align*}
\]

- Metric is not symmetric.
WHY THE XOR METRIC?

• why this metric?
  • 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

  $000100 \oplus 000110 = 000010$ common prefix $0001$, distance 2
  $010000 \oplus 000001 = 010001$ common prefix $0$, distance 17

• prefix matching guides the routing
Strictly related to the identifier tree

- assign a key $K$ to the node whose identifier is closer to $K$, according to the $\oplus$ metric
- key is assigned to a leaf 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 with respect to non symmetric distances (Chord fingers are not symmetric).

- unidirectional:
  - lookups for same key converge to the same path, and thus caching items along this path is good to avoid hotspots.
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

\[ 2^{i-1} \leq d(x,y) < 2^i \]

\[ \begin{align*}
X & = 010110 \\
Y & = 011110 \\
X \oplus Y & = 001000, \quad d(x,y) = 2^3 = 8 \text{ (minimal distance)}.
\end{align*} \]

\[ \begin{align*}
X & = 010110 \\
Y & = 011001 \\
X \oplus Y & = 001111, \quad d(x,y) = 2^4 - 1 = 15 \text{ (maximal distance)}
\end{align*} \]

• 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:

\[ 2^3 \leq d < 2^4 \]

0 1 1 1 ⊕ 1 0 0 0 = 1 1 1 1 = 15 (maximal, numerical distance is minimal)
0 1 1 1 ⊕ 1 1 1 1 = 1 0 0 0 = 8 (minimal, numerical difference is high)
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: 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 (not all the identifiers).

a leaf for each peer: the identifier of the leaf is a prefix of that of the peer.
A leaf identifier may be a prefix $P$ of the whole identifier of the peer if the overlay includes a single peer with the prefix $P$.

- 0011 uniquely identifies the red peer
- no other peer in the overlay with the same prefix
- this prefix allows to compute the distance
Routing table, the basic idea:
maintains some contacts (2 in the figure) for each subtree/common prefix
THE ROUTING TABLE

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

- for each row stores:
  
  (ID, IP, UDP port)

- 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
A COMPLETE EXAMPLE: KEYS AND R. TABLES

Subtrees for node N0

Subtrees for node N6

Subtrees for node N15

P: Prefix length
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
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:

<table>
<thead>
<tr>
<th>Tabella nodo 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</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
1. 4 risponde e viene spostato al fondo (6 è fuori)
K-BUCKETS MANAGEMENT

Esempio:

Tabella nodo 0

<table>
<thead>
<tr>
<th></th>
<th>[1,2)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>nodo1</td>
</tr>
<tr>
<td>1</td>
<td>[2,4)</td>
<td>nodo2</td>
</tr>
<tr>
<td>2</td>
<td>[4,0)</td>
<td>nodo4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nodo5</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
Why this politics for bucket management?
- x axis: minutes
- y axis: percentage of peers that, being on-line for \( x \) minutes, are online 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 uptime.
- the longer a node remains on-line, the higher is the probability that it remains online for an even longer interval of time
K BUCKETS MANAGEMENT

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 search that identifier
  - If the node with that identifier sends a reply it is inserted in the k-bucket
ROUTING AT A GLANCE

• $\alpha$: number of nodes to which the query is propagated, at each routing step

• Iterative Routing

• An example of routing for $\alpha=1$
ROUTING AT A GLANCE

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

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

Diagram:

- Step 3

Laura Ricci
Prefix Based DHT: Kademlia
Routing at a Glance

- **Black node**: query source (0011)
- **Orange Node**: query target (1110)
- **Green Node**: nodes known from 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.
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.
• 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.
Kademlia protocol consists of 4 primitive (not iterative) operations, defined as Remote Procedure Calls (RPCs) exploiting UDP:

- **PING** \( v \rightarrow w \)
  - probe node \( w \) to see if its online

- **STORE** \( v \rightarrow w \) (Key, Value)
  - instructs node \( w \) to store a \(<\text{key}, \text{value}>\) pair

- **FIND_NODE** \( v \rightarrow w(T) \)
  - In: \( T \), 160-bit ID
  - Out: the recipient of the RPC returns \( k \) contacts (\(<\text{IP}:\text{Port}, \text{NodeID}>\)) that it knows to be “closest” to \( T \)

- **FIND_VALUE** \( v \rightarrow w(T) \)
  - In: \( T \), 160-bit ID
  - Out: if a value corresponding to \( T \) is present, the associated data is returned, otherwise it is equivalent to FIND_NODE and a set of \( k \) triples is returned.
**NODE LOOK UP**

**select**  $k$-closest$= \alpha$ contacts from the non-empty $k$-bucket closest to the key

**if** there are fewer than $\alpha$ contacts in that bucket,

**select**  $k$-closest $= k$-closest $\cup$ closest contacts from other buckets.

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

**repeat**

**send** in parallel, asynchronously FIND_NODE (or FIND_VALUE) to the contacts in $k$-closest

each contact, if live, should normally return $k$ nodes, add them to $k$-closest

**add** to $k$-closest the new received nodes

**update** closestNode

**select** from $k$-closest $\alpha$ closest contacts which have not been queried yet and send them FIND_NODE

**until** no node closer to the target than closestNode is returned

**send** in parallel asynchronously FIND_NODE (or FIND_VALUE) to the $k$ closest nodes it has not already queried

**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 (Q)**

- P looks for the key Q (which is the identifier of a node or of a content)
  - looks in the bucket-list the $\alpha$ nodes closest to Q
  - looks in the k-bucket closest to the key and not empty. If it includes less than $\alpha$ nodes ($\alpha=3$), looks in close buckets
  - selected contacts may belong to different k-buckets

Here are stored the nodes closest to Q.
LOOK_UP (Q)

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 form P.
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
FindNode (Q)

Terminate when a round of FindNode(T) fails to return any closer nodes
to store a (key, value) pair, a participant locates the k closest nodes to the key and sends them STORE RPCs.

- data replicated on k 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 enter 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.
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
CHORD VERSUS KADEMLIA

- 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, neighborhood)