P2P Systems and Blockchains

Spring 2019,
instructor: Laura Ricci
laura.ricci@unipi.it

Lesson 3: THE CHORD DHT

01/03/2019
A Distributed Hash Table: Chord

- Motivation: more efficient search in P2P networks
- Idea: Hash tables offer key/value-mapping
- Principle: every peer and object has a unique name
  - Calculate a hash function on unique name
  - Peers and objects map onto points in hash-space
  - Peer “closest” to object is responsible for that object
- Many different research projects on this topic
  - Hash function, hash-space, and metric differ, but...
  - examples: Chord (MIT), CAN (UC Berkeley), Pastry (Microsoft & Rice Univ.), Tapestry (UC Berkeley)
- DHT as basis for building more complex services
LESSON OUTLINE

- Chord: general features
- The overlay topology: routing
- Self organization
  - node join
  - voluntary leave
  - faults
- Routing
  Modelling routing through Markov chains
- Analysis of the Chord topology through complex networks analysis tools (later)
Paper reference


developed in 2001 from a research group formed by researcher from MIT and University of California
CHORD: INTRODUCTION

• Based on a few, simple but powerful concepts:
  • easily to define and to implement
  • elegance
  • possibility of defining optimizations

• main characteristics:
  • applies consistent hashing concepts
  • routing to find the node managing a key
    • log(N) hops with high probability, where N is the total number of peers in the system
    • routing table size: log(N) with high probability
  • self-organization
  • self-adaptation in presence of new nodes joins and voluntary/abrupt nodes leave.
Application and Chord

- Application sits on top of Chord

- API has 4 functions:
  - **Join**: join the network
  - **Leave**: leave the network
  - **Store**: store a key-value pair
    - to store object with name “Foo” and value = 5
      - Calculate id = hash(“Foo”) (id often referred as key)
      - Send message STORE (id 5)
      - Message routed to node responsible for id
  - **Retrieve**: retrieve a stored value
CHORD: CONSISTENT HASHING

- each node (host) is paired with identifier $id$, obtained by SHA
  $id_{node} = SHA \text{(IP address, port)}$
- each data is paired with a unique identifier $k$ (or key), obtained by SHA-1
  $k = SHA \text{(data}_\text{name)}$
- Keys and nodes are mapped onto the same logical address space
A Distributed Hash Table: Chord

Laura Ricci

Dipartimento di Informatica
Università degli Studi di Pisa

CHORD: CONSISTENT HASHING

- hypothesis: consider identifiers (returned by SHA) of m bits, \([0, 2^m-1]\)
- define an ordering between the identifiers, based on their numerical value such that successor of \(2^m-1\) is 0 (modulo \(2^m\)): represented as a ring
- mapping algorithm (consistent hashing)
  - a key \(K\) is assigned to the first node \(n\) of the ring whose identifier is greater or equal (modulo the size of the ring) to \(K\).
  - \(n\) detected starting from \(K\) and following the ring clockwise
  - denoted \(\text{successor}(K)\)
LOOK UP?

- Assume a node look-ups a key K: how does the node find K?
- It depends from the information stored in the routing table.
- **Minimal Routing Table** of a node:
  - each node remembers only the next node on the ring (successor link)
  - this is the only knowledge of a node about other nodes of the ring
LOOK UP: AN INEFFICIENT SOLUTION

- A simple routing algorithm:
- each node remembers only next node on the ring (successor link)
  - send the query with $key=x$ to the successor until $n=\text{successor}(x)$ is detected
  - $n$ returns the query results
- Advantages:
  - simple
  - routing tables $O(1)$
- Disadvantages:
  - routing $O(\frac{1}{2} \times n)$, linear
  - a single node crash breaks the ring
LOOK UP: AN INEFFICIENT SOLUTION

- A simple routing algorithm, each node:
  - has a single link towards its successor
  - sends the key=x to its successor until it finds it does not find n=successor(x)
  - n returns the query results

Advantages:
- simple
- routing table O(1)

Disadvantages:
- Routing O(½ * n), linear
- a single node crash breaks the ring
LOOK UP: AN INEFFICIENT SOLUTION

- A simple routing algorithm, each node:
  - has a single link towards its successor
  - sends the key=x to its successor until it finds it does not find n=successor(x)
  - n returns the query results

- Advantages:
  - simple
  - routing table O(1)

- Disadvantages:
  - routing O(½ * n), linear
  - a single node crash breaks the ring
LOOK UP: AN INEFFICIENT SOLUTION

- A simple routing algorithm, each node:
  - has a single link towards its successor
  - sends the key=\(x\) to its successor until it finds it does not find \(n=\text{successor}(x)\)
  - \(n\) returns the query results

- Advantages:
  - simple
  - routing table \(O(1)\)

- Disadvantages:
  - Routing \(O(\frac{1}{2} \times n)\), linear
  - a single node crash breaks the ring
A distributed algorithm: use Remote Procedure Calls (RPC) to code it

- (a, b] the segment of the ring moving clockwise from but not including a until and including b.
- n.foo( ) denotes an RPC of foo( )@node n.
- n.bar denotes an RPC to fetch the value of the variable bar in node n.
- Hypothesis: a node has also a pointer to its predecessor.
AN INEFFICIENT SOLUTION: PUT AND GETS

```java
procedure n.put(id, value) {
    s = findSuccessor(id)
    s.store(id, value)
}
```

```java
procedure n.get(id) {
    s = findSuccessor(id)
    return s.retrieve(id)
}
```

PUT and GET are nothing but lookups!
CHORD ROUTING: REDUCING NUMBER OF STEPS

- each node has links towards $z$ neighbours
- If $z = n$ the overlay is a complete mesh
  - routing hops: $O(1)$
  - size of the routing tables: $O(n)$, limited scalability
- A compromise: a logarithmic mesh of the nodes of the ring
- each node stores several links towards some close neighbours (on the ring) and a few links towards far neighbours
  - a limited number of links for each node
  - routing is more accurate in the neighbourhood of a node, more approximate towards far neighbours
- routing algorithm:
  - send a query for the key $k$ to the farthest known predecessor of $k$
CHORD ROUTING: REDUCING THE COMPLEXITY

- Let us define the **distance** between two identifier $I_1$ and $I_2$ of the Chord ring as the number of identifiers between $I_1$ and $I_2$

- Basic idea for building the routing table (**finger table**)
  - the node $n$ with identifier $x$ in the ring knows a set of **at most $m$ nodes** (for a ring of $m$ bits) and the distance from these nodes **exponentially increases**
  - each node has “fingers” to nodes $\frac{1}{2}$ way around the ID space from it, $\frac{1}{4}$ the way.
  - $\text{finger}[i]$ at $p$ contains $\text{successor}(p+2^{i-1})$
  - $\text{successor}$ is $\text{finger}[1]$
  - the number of different entries in the finger table of a node $n$ is generally is less than $m$
Chord Overlay:
a logarithmic mesh of the nodes of the ring
Each node owns a **finger table** (routing table)

- if \( m \) is the number of bits exploited for the identifiers, the table includes at most \( m \) links to Chord nodes
- at node \( n \): the entry \( \text{finger}[i] \) is a pointer to \( \text{successor}(n + 2^{i-1}) \), \( 1 \leq i \leq m \)

**Data structure of Node 0**

<table>
<thead>
<tr>
<th>( i )</th>
<th>target</th>
<th>link.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Number of keys: 6
Each node owns a **finger table** (routing table)

- if \( m \) is the number of bits exploited for the identifiers, the table includes **at most** \( m \) links to Chord nodes
- at node \( n \): the entry \( \text{finger}[i] \) is a pointer to \( \text{successor}(n + 2^{i-1}) \), \( 1 \leq i \leq m \)

**Data Structure of Node 0**

**Data Structure of Node 1**
Each node owns a finger table (routing table)

- if $m$ is the number of bits exploited for the identifiers, the table includes at most $m$ links to Chord nodes
- at node $n$: the entry $\text{finger}[i]$ is a pointer to $\text{successor}(n + 2^{i-1})$, $1 \leq i \leq m$

Data Structure of Node 0

<table>
<thead>
<tr>
<th>$i$</th>
<th>target</th>
<th>link.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Data Structure of Node 1

<table>
<thead>
<tr>
<th>$i$</th>
<th>target</th>
<th>link.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Data Structure of Node 3

<table>
<thead>
<tr>
<th>$i$</th>
<th>target</th>
<th>link.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
The data structures of a node

Each node maintains some data structures to support routing

1) the finger table:
   - has at most $m$ entries (for instance $m=160$)
   - For small $I$ the fingers entries are the same: store only different entries
   - actually, the size of finger table is logarithmic with respect to the number of nodes of the overlay

2) a link to the successor and to the predecessor on the ring

3) of course the node maintains also a set of keys
LOOK UP IN CHORD

- **Routing Protocol**: each node propagates a query with key $k$ to the farthest finger preceding $k$, by considering the clockwise ordering.

- The propagation of the key goes on until the node $n$ such that:
  $$((n < k) \text{ and } \text{successor}(n) \geq k) \text{ (arithmetic modulo } 2^m)$$

  in this case $\text{successor}(n)$ owns the key.

### Table: Lookup (44)

<table>
<thead>
<tr>
<th>$i$</th>
<th>$2^i-1$</th>
<th>Target</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>20</td>
<td>23</td>
</tr>
</tbody>
</table>
LOOK UP IN CHORD

- **Routing Protocol**: each node propagates a query with key \( k \) to the farthest finger preceding \( k \), by considering the clockwise ordering.
- The propagation of the key goes on until the node \( n \) such that:
  \[
  ((n < k) \text{ and } \text{successor}(n) \geq k) \text{ (arithmetic modulo } 2^m \text{)}
  \]
in this case \( \text{successor}(n) \) owns the key.

### lookup (44)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( 2^i-1 )</th>
<th>Target</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>20</td>
<td>23</td>
</tr>
</tbody>
</table>
LOOK UP IN CHORD

- **Routing Protocol**: each node propagates a query with key k to the farthest finger preceding k, by considering the clockwise ordering.
- The propagation of the key goes on until the node n such that:
  
  $((n < k) \text{ and } \text{successor}(n) \geq k) \text{ (arithmetic modulo } 2^m)$

  
  In this case successor(n) owns the key.

lookup (44)

<table>
<thead>
<tr>
<th>i</th>
<th>$2^{i-1}$</th>
<th>Target</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>31</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>55</td>
<td>56</td>
</tr>
</tbody>
</table>
LOOK UP IN CHORD

- **Routing Protocol**: each node propagates a query with key $k$ to the farthest finger preceding $k$, by considering the clockwise ordering.
- The propagation of the key goes on until the node $n$ such that:
  $$(n < k) \text{ and } \text{successor}(n) \geq k$$ (arithmetic modulo $2^m$)

In this case, $\text{successor}(n)$ owns the key.
LOOK UP IN CHORD

- **Routing Protocol**: each node propagates a query with key \( k \) to the farthest finger preceding \( k \), by considering the clockwise ordering.
- The propagation of the key goes on until the node \( n \) such that:

  \[(n < k) \text{ and } \text{successor}(n) \geq k\]  

( arithmetic modulo \( 2^m \))

in this case \( \text{successor}(n) \) owns the key.

### lookup (44)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( 2^i - 1 )</th>
<th>Target</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>40</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>41</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>43</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
LOOK UP IN CHORD

- **Routing Protocol**: each node propagates a query with key k to the farthest finger preceding k, by considering the clockwise ordering.
- The propagation of the key goes on until the node n such that:
  \[ (n < k) \text{ and } \text{successor}(n) \geq k \] (arithmetic modulo \(2^m\))

In this case, \(\text{successor}(n)\) owns the key.

<table>
<thead>
<tr>
<th>i</th>
<th>(2^{i-1})</th>
<th>Target</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>40</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>41</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>43</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
• **Routing Protocol**: each node propagates a query with key $k$ to the farthest finger preceding $k$, by considering the clockwise ordering.

• The propagation of the key goes on until the node $n$ such that:

  $((n < k) \text{ and } \text{successor}(n) \geq k) \text{ (arithmetic modulo } 2^m\text{)}$

in this case $\text{successor}(n)$ owns the key.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$2^{i-1}$</th>
<th>Target</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>43</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>44</td>
<td>45</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>46</td>
<td>49</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>50</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>58</td>
<td>60</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>
• Routing Protocol: each node propagates a query with key $k$ to the farthest finger preceding $k$, by considering the clockwise ordering.

• The propagation of the key goes on until the node $n$ such that:

$$(n < k) \text{ and } \text{successor}(n) \geq k \text{ (arithmetic modulo } 2^m)$$

in this case $\text{successor}(n)$ owns the key.

**LOOK UP IN CHORD**
// ask node n to find the successor of id
procedure n.findSuccessor(id) {
    if (predecessor $\neq$ nil and id $\in$ (predecessor, n]) then return n
    else if (id $\in$ (n, successor]) then
        return successor
    else // forward the query around the circle
        return successor.findSuccessor(id)
}

closestPrecedingNode(id)
// ask node n to find the successor of id
procedure n.findSuccessor(id) {
    if (predecessor ≠ nil and id ∈ (predecessor, n)) then return n
else if (id ∈ (n, successor)) then
    return successor
else { // forward the query around the circle
    m := closestPrecedingNode(id)
    return m.findSuccessor(id)
}
}

// search locally for the highest predecessor of id
procedure closestPrecedingNode(id) {
    for i = m downto 1 do {
        if (finger[i] ∈ (n, id)) then
            return finger[i]
    }
    return n
}
CHORD: HANDLING CHURN

- what about joins and failures/leaves?
  - nodes come and go as they wish.

- what about data?
  - should I lose my doc because some kid decided to shut down his/her machine and he/she happened to store my file?

- so actually we just started....we have to consider
  - node join
  - node voluntary leave
  - node crash
  
  and try to maintain the ring connected....
CHORD: NODE JOIN

1. The new node $n$ computes a SHA-1 identifier $ID$
2. contacts a bootstrap node, and links to its successor
3. builds its own finger table (may be lazy)
4. receives the keys to manage
5. periodically: node predecessor execute a stabilization procedure
6. periodically all nodes stabilize finger tables for handling churn

ID = Hash( ) = 6
2. contacts the bootstrap node, and links to its successor
   - remind: a key based routing detects the successor of the key
   - the new node (6) sends to the bootstrap node (1) a query with key equal to its identifier (6)
     - key-based routing detects the successor of the key 6 on the ring
     - in our case: successor of node 6 is node 0
   - in this way node 6 gains knowledge of its successor on the ring

Key=6
ID = SHA() = 6
1. The new node \( n \) computes a SHA-1 identifier \( ID \)
2. contacts a bootstrap node, and \( \text{links to its successor} \)
3. builds its own finger table (may be lazy)
4. receives the keys to manage
5. periodically: node predecessor execute a stabilization procedure
6. periodically finger table stabilization of all the nodes

\[ \text{finger table} \]

\[
\begin{array}{|c|c|c|}
\hline
i & \text{target} & \text{link} \\
\hline
1 & 1 & 1 \\
2 & 2 & 3 \\
3 & 4 & 0 \\
\hline
\end{array}
\]

\[ \text{keys} \]

\[ 6 \]

\[ \text{finger table} \]

\[
\begin{array}{|c|c|c|}
\hline
i & \text{target} & \text{link} \\
\hline
1 & 2 & 3 \\
2 & 3 & 3 \\
3 & 5 & 0 \\
\hline
\end{array}
\]

\[ \text{keys} \]

\[ 1 \]

\[ \text{finger table} \]

\[
\begin{array}{|c|c|c|}
\hline
i & \text{target} & \text{link} \\
\hline
1 & 4 & 0 \\
2 & 5 & 0 \\
3 & 7 & 0 \\
\hline
\end{array}
\]

\[ \text{keys} \]

\[ 2 \]

1. \( ID = \text{SHA}(\ldots) = 6 \)

Key = 6
3. builds its own finger table (but be lazy)

- define a structure with m entries. For each row:
- locate the target for row i: $n + 2^{i-1}$
- invoke \texttt{findSuccessor}(n + 2^{i-1}) (finger[i] points to successor(n + 2^{i-1}), 1 \leq i \leq m)
- again, exploit \texttt{findSuccessor} to find the successor of the i-th target

![Diagram of Chord: Node Join]

<table>
<thead>
<tr>
<th>Finger Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

Key = 7
3. builds its own finger table (may be lazy)

- define a structure with \( m \) entries. For each row:
- locate the target for row \( i: n + 2^{i-1} \)
- invoke \( \text{findSuccessor}(n + 2^{i-1}) \) (\( \text{finger}[i] \) points to \( \text{successor}(n + 2^{i-1}), \ 1 \leq i \leq m \))
- again, exploit \( \text{findSuccessor} \) to find the successor of the \( i \)-th target
3. builds its own finger table (may be lazy)

- define a structure with m entries. For each row:
- locate the target for row i: n + 2^{i-1}
- invoke findSuccessor(n + 2^{i-1}) (finger[i] points to successor(n + 2^{i-1}), 1 ≤ i ≤ m)
  - again, exploit findSuccessor to find the successor of the i-th target
CHORD: NODE JOIN

3. builds its own finger table (may be lazy)
   • define a structure with m entries. For each row:
   • locate the target for row i: n + 2^{i-1}
   • invoke findSuccessor(n + 2^{i-1}) (finger[i] points to successor(n + 2^{i-1}), 1 \leq i \leq m)
     • again, exploit findSuccessor to find the successor of the i-th target
3. builds its own finger table (may be lazy)
   - define a structure with m entries. For each row:
   - locate the target for row i: n + 2^{i-1}
   - invoke findSuccessor(n + 2^{i-1}) (finger[i] points to successor(n + 2^{i-1}), 1 \leq i \leq m)
     - again, exploit findSuccessor to find the successor of the i-th target
1. The new node $n$ computes a SHA-1 identifier $ID$
2. contacts the bootstrap node, and links to its successor
3. builds its own finger table (may be lazy)
4. receives the keys to manage
5. periodically: node predecessor execute a stabilization procedure
6. periodically: finger table stabilization of all the nodes

1. $ID = \text{rand()} = 6$

Key = 6

finger table

<table>
<thead>
<tr>
<th>$i$</th>
<th>target</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

keys

<table>
<thead>
<tr>
<th>$i$</th>
<th>target</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

finger table

<table>
<thead>
<tr>
<th>$i$</th>
<th>target</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

keys

<table>
<thead>
<tr>
<th>$i$</th>
<th>target</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>
CHORD: NODE JOIN

1. the new node $n$ computes a SHA-1 identifier ID
2. contacts the bootstrap node, and links to its successor
3. builds its own finger table (may be lazy)
4. receives the keys to manage
5. periodically: node predecessor execute a stabilization procedure
6. periodically all nodes stabilize finger tables for handling churn
4. receives the keys to manage
- to respect the mapping keys-nodes, the node 6 must receive all the keys less or equal to 6 from node 0.
- this has to be implemented by the application running on the DHT
CHORD: RING MAINTENANCE

- all these operations take time to complete

- what does it happen during a node join?
  - finger tables may become inconsistent because of the join of the new node
  - the remaining nodes goes on searching content
  - we would like to avoid false negatives, because of this inconsistency
  - nodes may join the ring concurrently
in addition to the successor pointer (at least one successor, but may be more than one), every node has a **predecessor pointer** as well for ring maintenance.

- predecessor of node $n$ is the first node met in **anticlockwise direction** starting at $n-1$

- These pointers make it possible to find a content even if the finger tables are not updated.

- The correctness of the protocol depends on the consistency of all the successor pointers.
When a node joins the ring, it initializes only its successor and notifies its presence to its successor.

- the correctness of the look up depends on the correctness of the successor of each node of the Chord ring.

- Periodically, not immediately:
  - also successor pointers of other nodes are stabilized
    - the stabilize() procedure has been executed: it stabilizes all the successor and predecessor pointers
  - build the content of the finger table (not all together at join time as shown before) and update the other finger tables
    - the fixfingers() procedure has been executed both from the joining node and from other nodes of the overlay: it stabilizes the fingers in the finger tables
CHORD: LAZY OPERATIONS

- **the stabilize( ) procedure**
  - instead of contacting the predecessor immediately, the stabilize routine is called periodically
  - asks successor who its predecessor is
  - if it is me, we’re good
  - if not, then we have a new successor, and we notify him of us
  - it sets its predecessor correctly

- **the fixfingers( ) procedure**
  - instead of actively initializing the entire finger table, periodically set some random finger in the finger table and fix it
  - the same for the other nodes
• Before a new join:
  • the ring is in a stable state: successor and predecessor pointers are consistent
CHORD: LINKING TO THE SUCCESSOR

- Node’s successor pointer should be up to date
  - for correctly executing lookups
  - the new node (13)
    - finds its successor on the ring
      - how? start at a bootstrap node, send a query `findSuccessor()` with key the identifier of the node itself
    - joins the ring linking to its successor
  - may now inform its successor of its presence, and the successor sets it as its predecessor
CHORD: LINKING TO THE PREDECESSOR

• The new node (13) is now correctly linked to its successor (and the successor to its predecessor)
• but its predecessor (11) does not know it!
• this means that the other nodes of the ring may not be aware of its presence
  • no problem until the keys are transferred to it
  • if its has no key, no look up regards that node
  • it can execute look-up correctly
CHORD: STABILIZE

• why the successor of the joining node does not notify immediately its previous predecessor about the existence of the new node?

• look at the following scenario: node N26 is the joining node
  • N21 must point to N26 only after N26 has received the keys
  • N21 does not point to N26 when N26 has found its successor, but later, through the stabilize procedure
CHORD: STABILIZE

- each node periodically executes a ring stabilization procedure

// Periodically at n:
\[ v := \text{succ}\text{.pred} \]
\[ \text{if } (v \neq \text{nil and } v \in (n, \text{succ})) \text{ then} \]
\[ \text{set succ } := v \]
\[ \text{send a notify}(n) \text{ to succ} \]

- if the current predecessor of the successor is different from itself set the current successor to current predecessor of the successor
CHORD: STABILIZE

- When the new node receives the notification, it sets its predecessor at the value notified.
- The stabilization procedure is now complete.

// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
set pred := p
CHORD: FIX FINGERS

- finger tables of other nodes (the joining node builds its finger table) become inconsistent, because of the join of new nodes
- periodical stabilization of the finger tables
- at each iteration fix a finger (or choose a finger at random)
  - submit a query with as target that finger

```c
// When receiving notify(p) at n:
procedure n.fixFingers() {
    next := next + 1
    if (next > m) then
        next := 1
    finger[next] := findSuccessor(n ⊕ 2^(next - 1))
}
```
CHORD: FIX FINGER

- \text{Succ}(N48) = N60
- \text{finger 6 of } N21
  - \text{Succ}(21 + 2^{(6-1)}) = \text{Succ}(53) = N60.$
CHORD: FIX FINGERS

• N56 joins the overlay and links to its successor
  • Finger 6 of node N21 is no more correct!

• N21 tries to fix finger 6 by submitting a query for key 53.
  • \( \text{Succ}(21 + 2^{(6-1)}) = \text{Succ}(53) = \cdots \)

• If the ring is not yet stabilized, N48 has not fixed its successor, so the finger cannot be fixed
• Ring maintenance is executed
• N48 stabilizes its successor (points to N56)
At the next attempt of N21 to fix its finger Finger 6, the reply of N48 is N56 correct and N21 may correct its finger

• \( \text{Succ}(21 + 2^{(6-1)}) = \text{Succ}(53) = N56 \)
When a search is initiated before the system is in stable state, after a node join:

- **case 1**: not all the pointers to the successor nodes have been stabilized and some keys have been transferred. The search may fail and has to be retried later. (correctness issue: false negatives)

- **case 2**: each node has updated the pointer to its real successor in the ring and pairs key/data are correctly transferred between the nodes, but the finger tables have not been completely updated. No false negative, but the search may be slowed down (efficiency issue)

- **case 3**: all the finger tables are “reasonably updated”, the routing requires $O(\log N)$ steps
INCREASING THE ROBUSTNESS

- the correctness of the routing algorithm is guaranteed if each node maintains updated the reference to its real successor node on the ring even in case of multiple simultaneous failures

- A node has a **successors list** of size \( r \) containing the immediate \( r \) successors
  
  \[
  \text{succ}(n+1) \\
  \text{succ(succ}(n+1)+1) \\
  \text{succ(succ(succ}(n+1)+1)+1)
  \]

- higher value of \( r \) implies a greater robustness of the system

- what is a good value for \( r \)? \( \log(N) \)

- the list is maintained consistent through a stabilization procedure
HANDLING FAILURES: RING MAINTENANCE

- node 13, leaves the overlay abruptly
HANDLING FAILURES: RING MAINTENANCE

- node 13, leaves the overlay abruptly
- two dangling references
HANDLING FAILURES: RING MAINTENANCE

- node 13, leaves the overlay abruptly
- when 15 detects the fault, it sets pred to nil
HANDLING FAILURES: RING MAINTENANCE

- node 13, leaves the overlay abruptly
- when 15 detects the fault, it sets pred to nil
- when 11, detects the fault, it sets succ to the first “live” successor in the successor list and notifies this node

![Diagram showing ring maintenance process]
• node 15 update its prec field to 11
Each communication with the fingers is controlled through time outs. If a time out expires:

- the query is sent to the previous finger, to avoid crossing the target node
- the crashed finger is replaced by its successor in the Finger table (trigger repair) (remember finger[i] points to successor(n + 2^{i-1}), 1 ≤ i ≤ m)

Example: node 39 fails, node 23 must 'repair' its finger table which includes a pointer to 39
Each communication with the fingers is controlled through time outs. If a time out expires:

- the query is sent to the previous finger, to avoid crossing the target node
- the crashed finger is replaced by its successor in the Finger table (trigger repair) (remember finger[i] points to successor(n + 2^i-1), 1 ≤ i ≤ m)

**An example:**
Node 23 substitutes in its finger table the link 39 with the previous finger

<table>
<thead>
<tr>
<th>i</th>
<th>target</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>39</td>
<td>56</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>56</td>
</tr>
</tbody>
</table>

Key 44 is sent to 33 instead of 39.
An inconsistent scenario: a node loses a reference to its real successor

- The next three successors of node 23 (26,30,33) fail.
- The real successor of node 23 becomes node 37.
- But, since node 23 has not a reference to node 37 in its finger table, 23 considers 39 as its new successor.
- The key 35 is sent to node 39, while this key is managed by node 37.
CHORD: DATA REPLICATION

- Chord does not guarantee that data are not lost when failure occurs.
- But... Chord defines mechanisms to guarantee the reliability.
- Each application exploiting the Chord level may exploit the successor list to guarantee that a data is replicated onto the $r$ successors of a node.
- When a node fails, the system may exploit the replica for finding the required data.
CHORD: NODES VOLUNTARY DEPARTURE

• Voluntary departure of a node
  • voluntary shut-down versus node failure: in the simplest case, it is managed like a fault

• Optimization: the node n which is going to leave the ring
  • notifies this to the successor, the predecessor and to the fingers
    • the predecessor may remove n from the successors list
    • the predecessor may add to its successors list the first node in the successor list of n
  • transfers its keys to the successor on the ring
CHORD: PROJECT CHOICES

A minimal approach

• Chord manages the key-based routing
• the management of the data is a task of the applications
  • persistence
  • consistency
  • fairness

A soft approach

• nodes delete the pairs \((key, value)\) after a given interval of time of time (refresh time) from the last insertion.
• the applications run a periodic refreshment of the pairs \((key, value)\)
• in this way the new nodes acquire the data
• if a node fails, the application waits for the refresh interval to have data again available
CHORD: SIMULATION RESULTS

- overlay = $2^k$ nodes, $100 \times 2^k$ keys, $3 < k < 14$.
- for each value of $k$, we consider a set of keys chosen at random and a simulation is run
- for each key, evaluate the hops number for the look-up
- Lookup Path Length $\sim \frac{1}{2} \log_2(N)$
- theoretical results are confirmed
- the figure shows the the average and the variance (logarithmic scale)
Average paths length $\approx 6 = \frac{1}{2} \log_2 (2^{12})$
CHORD: THE PROTOTYPE

- Development of a Chord prototype developed on the Internet
- Chord nodes located at 10 sites (located in different USA states)
- Different experiments varying the number of nodes: for each number of nodes, 16 queries for keys chosen at random
- Average latency varies from 180 ms. to 300 ms, depending on the number of nodes
CHORD: PERFORMANCE

Scalability
Low impact of the number of nodes on the latency
• Chord dynamically manages the network changes
  • node failures
  • network failure
  • new nodes arrival
  • voluntary leaves of nodes

• Problem: maintaining a consistent system state in presence of dynamical changes
  • updating information required for routing messages
  • routing correctness: each node maintains its real successor on the ring up to date
  • routing efficiency: it depends from the prompt update of the Finger Tables
  • failure tolerance
CHORD: CONCLUSIONS

● Complexity
  • look-up messages: $O(\log N)$ hops
  • memory for each node: $O(\log N)$
  • messages for self organization (join/leave/fail): $O(\log^2 N)$

● Advantage
  • theoretical models and complexity proofs
  • simple and flexible

● Disadvantages
  • physical proximity is not considered
  • real scenarios: disadvantage scenarios may occur

● Optimizations
  • e.g. proximity, bi-directional links, load balancing, etc.
  • applications: to be seen in the next lessons