Lesson 6
CHORD: HANDLING DYNAMISM
RING MAINTENANCE

Laura Ricci
10/03/2017
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 machine and he 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....
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

\[ ID = \text{Hash}( ) = 6 \]
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 = SHA(.... ) = 6
2. Key = 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
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

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

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

$$\begin{array}{|c|c|c|}
\hline
i & \text{target} & \text{link} \\
\hline
1 & 4 & 0 \\
2 & 5 & 0 \\
3 & 7 & 0 \\
\hline
\end{array}$$
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 \textit{findSuccessor}(n + 2^{i-1}) \texttt{(finger[i] points to successor(n + 2^{i-1}), 1 \leq i \leq m)}
   • again, exploit \textit{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
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 \( \text{findSuccessor}(n + 2^{i-1}) \) (\( \text{finger}[i] \) 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($n + 2^{i-1}$), $1 \leq i \leq m$)
  * again, exploit findSuccessor to find the successor of the i-th target
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 finger table stabilization of all the nodes
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

• Lazy join:
  • when a node joins the ring, it initializes only its successor and notifies this to its successor
  • periodically, it refreshes the content of its finger table
  • periodically all successor and predecessor pointers are stabilized

• The correctness of the look up depends on the correctness of the successor of each node of the Chord ring

• All the links of a node are correctly updated when
  • the stabilize() procedure has been executed: it stabilizes all the successor and predecessor pointers
  • the fixfingers() procedure has been executed both from the joining node and from other nodes of the overlay: it stabilizes each fingers in the finger tables
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 go on searching content

we would like to avoid false negatives, because of this inconsistency

nodes may join the ring concurrently
CHORD: RING MAINTENANCE

- 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 \( n-1 \)

- **Periodic stabilization** is used to make pointers eventually correct.
  - try pointing \texttt{succ} to closest alive successor.
  - try pointing \texttt{pred} to closest alive predecessor.

- This 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 the successor pointers.
CHORD: RING MAINTENANCE

- Lazy join:
  - when a node joins the ring, it initializes only its successor and notifies its presence to its successor
  - Periodically, not immediately:
    - it refreshes the content of its finger table
    - all successor and predecessor pointers are stabilized

- all the links of a node are correctly updated when
  - the stabilize( ) procedure has been executed: it stabilizes all the successor and predecessor pointers
  - the fixfingers( ) procedure has been executed both from the joining node and from other nodes of the overlay: it stabilizes all fingers in the finger tables

- The correctness of the lookup depends on the correctness of the successor of each node of the Chord ring
CHORD: RING MAINTENANCE

- Before a new join:
  - the ring is in a stable state: successor and predecessor pointers are consistent
CHORD: RING MAINTENANCE

- 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
CHORD: RING MAINTENANCE

- The new node is now correctly linked to its successor
- It is not linked to its predecessor
- This means that the other nodes of the ring are not aware of its presence
  - no problem until the keys are transferred to it
CHORD: RING MAINTENANCE

- each node periodically executes a ring stabilization procedure

```plaintext
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n, succ]) then
    set succ := v
    send a notify(n) to succ
```

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

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

• When the new node receives the notification, it sets its predecessor at the value notified
CHORD: FINGERS STABILIZATION

- finger tables become inconsistent, because of new joins.
- periodically stabilization of the finger tables
- at each iteration fix a finger
  - submit a query with the target of 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 \oplus 2^{(next - 1)})
}
```
CHORD: FIX FINGER TABLES

- \( \text{Succ}(N_{48}) = N_{60} \)
- \( \text{finger 6 of } N_{21} \)
  - \( \text{Succ}(21 + 2^{(6-1)}) = \text{Succ}(53) = N_{60}. \)
CHORD: FIX FINGER TABLES

- 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.
  - \( \text{Succ}(21 + 2^{(6-1)}) = \text{Succ}(53) = ??? \)

- If the ring is not yet stabilized, N48 has not fixed its successor, so the finger cannot be fixed
CHORD: FIX FINGERS

- 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

- \( \text{Succ}(21 + 2^{(6-1)}) = \text{Succ}(53) = N56 \)
CHORD: INCONSISTENCIES

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.

• **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.

• **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

- anode has a successors list of size $r$ containing the immediate $r$ successors
  
  $\text{succ}(n+1)$
  $\text{succ}(\text{succ}(n+1)+1)$
  $\text{succ}(\text{succ}(\text{succ}(n+1)+1)+1)$

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

- node 13, leaves the overlay abruptly
- two dangling references
• 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
- 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 \( \text{finger}[i] \) points to successor \( n + 2^{i-1} \), \( 1 \leq i \leq m \))

**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 $\text{finger}[i]$ points to successor($n + 2^{i-1}$), $1 \leq i \leq m$)

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

Key 44 is sent to 33 instead of 39

<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>
FINGER TABLES: INCONSISTENCIES

An inconsistent scenario: a node loses a reference to its real successor

- The next three successors of node 23 (26,30,33) fail.
- The successor of node 23 becomes node 37.
- 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.

```
lookup (35)
```

```
<table>
<thead>
<tr>
<th></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>39</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>56</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>target</th>
<th>link</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td>39</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>39</td>
</tr>
<tr>
<td>5</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>56</td>
</tr>
</tbody>
</table>
```
Chord: handling churn
Laura Ricci

Dipartimento di Informatica
Università degli Studi di Pisa

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
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 (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 1st 99th percentile and the average (logarithmic scale).
**CHORD: SIMULATION RESULTS**

Average paths length $\approx 6 = \frac{1}{2} \log_2 (2^{12})$

Path Length: PDF (Probability Density Function) for an overlay of $2^{12}$ nodes
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: AUTO ORGANIZATION

- 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): $O(\log^2 N)$

• Advantages
  • 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