Peer to Peer and Overlay Networks
Structured P2P networks: Chord

Ricardo Lopes Pereira
ricardo.pereira@inesc-id.pt
IST
10-10-2011

1 Images from *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications* by Ion Stoica et al
1 Motivation

2 Structured networks
   • Characteristics

3 Chord
   • Characteristics
   • Operation
   • Routing
   • Churn
Limitations of unstructured networks

- Non deterministic searches:
  - Rare objects are difficult to locate
  - False negatives
- Traffic is high and difficult to estimate (without knowing topology and search pattern)
- Search effort (consumed resources) is unknown beforehand (no reasonable upper or lower bounds)
- Not scalable
Characteristics

- Usually provide a flexible service for building other applications
- Each node has a unique identifier
- Each object has unique identifier
- Each node is responsible for storing objects whose identifiers lay on its sphere of influence
- Typically, identifier space is the same for objects and nodes
- Load balancing is intrinsic property
- Operational semantic similar to that of an HashTable: insert object (key, object), search object (key), remove object (key)
- DHT: distributed hash table
- Provide the bases for more complex P2P applications
Advantages

- Deterministic properties:
  - Will find a object if it exists
  - Number of hops for a search is bounded (function of the network size)

- Very scalable
Limitations

- Search by object identifier only
  - Search using expressions not possible
  - Search using boolean operations is not possible
  - Search over various parameters is not possible
Characteristics

- Node’s position in the network is determined by an algorithm
- Searches: forwarding decision is taken locally at each node
- Searches: each hop takes us closer to the final node
- Protocols take peer churn into account
- Some protocols take into account some of the characteristics of the network (e.g.: topology or delay)
Geometry and forwarding

- Selected geometry defines: available routes for messages
- Forwarding: selects, among the available paths, the ones to use
- Geometry selection must take into account:
  - The maintenance cost due to peer churn
  - Operation under high and low peer density
- Goals: limit forwarding delay in the average and worst cases
Characteristics

- It only maps keys into nodes
- If one stores key/value pairs in mapped node, it allows search/retrieval by key
- In a stable membership state, each node:
  - Keeps state about $O(\log N)$ other nodes
  - Finds the node responsible for a key in $O(\log N)$ hops
- Each entry or departure results, with high probability, in $O(\log^2 N)$ update messages
Properties

- Unidimensional identifier space (circle)
- Load balancing: keys are uniformly distributed
- Decentralized: all nodes are equal
- Scalable: retrieval cost is $O(\log N)$
- Available: it can withstand a constantly changing set of nodes (entries and departures). Node responsible for the key is always found (content might not)
- Performance degrades smoothly when peers are unable to maintain $O(\log N)$ updated routing entries
- Flexible name structure. Only maps ID (hash) to node
- Facilitates the validation of data authenticity: just use a cryptographic hash function of the data to generate the key
motivation
structured networks
chord
characteristics
operation
routing
churn
how it’s used

- Software library which provides two functions:
  - Given a key, returns the IP/port of the node responsible for the key
  - Callback function for notifying the applications about changes in the identifier interval for which the node is responsible

- It’s up to the application to implement most services, such as: authentication, caching, replication, ...
Operation

- Uses a cryptographic hash function (SHA-1) to place each node (IP)/key in a $m$ bit unidimensional space
- $m$ is large enough for the probability of collisions to be negligible
- Each key is the responsibility of the node with equal or immediately superior ID
Successor: assured localization

- Each node maintains a reference to its successor in the circle
- This ensures that we can find any node (the responsible for a key) in $O(N)$
- It ensures that it is possible for a node to find its successor (the first node with an ID superior to its)
Each node maintains up to $m$ routing entries (IP, porto, id).

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$finger[k].start$</td>
<td>$(n + 2^{k-1}) \mod 2^m$, $1 \leq k \leq m$</td>
</tr>
<tr>
<td>.interval</td>
<td>$[finger[k].start, finger[k+1].start)$</td>
</tr>
<tr>
<td>.node</td>
<td>first node $\geq n.finger[k].start$</td>
</tr>
<tr>
<td>successor</td>
<td>the next node on the identifier circle; $finger[1].node$</td>
</tr>
<tr>
<td>predecessor</td>
<td>the previous node on the identifier circle</td>
</tr>
</tbody>
</table>
Information density diminishes with distance. Each node knows more about the nodes closest to itself.
Forwarding

// ask node n to find id’s successor
nₜ.ﬁnd_successor(id)
  n’ = find_predecessor(id);
  return n’.successor;

// ask node n to find id’s predecessor
n.ﬁnd_predecessor(id)
  n’ = n;
  while (id ∉ (n’, n’.successor])
    n’ = n’.closest_preceding_finger(id);
  return n’;

// return closest finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if (ﬁnger[i].node ∈ (n, id))
      return ﬁnger[i].node;
  return n;

• Node may not know destiny, but will know a closer node
• Uses finger which is the largest predecessor of the key
• It cannot use the interval where the key falls, for there might have been changes in the network membership (only successor is guaranteed)
• At each step, distance is cut in half
After a node enters it must be ensured that:
- The successor of each node is correct
- Each key $k$ is under the responsibility of $\text{successor}(k)$

In order to maintain performance, it is desirable that the finger tables be correct.
Question

Which information must be updated when a node enters the network?
Non simultaneous entries

- Nodes maintain information on its predecessor, to aid in the entry process.
- When a node enters, it is necessary to:
  - Define the predecessor and the fingers of the new node. $O(m\log N)$ reduces to $O(\log^2 N)$ due to empty intervals which cause fingers to be repeated.
  - Update the predecessors and fingers of the other nodes, to reflect the entry of the new node. $O(\log^2 N)$
  - Notify the above layer (application) to transfer, to the new node from its successor, the data associated with the keys for which it is responsible.
- Node joins the network by communicating with any node which already is part of the network.
Non simultaneous entries
Non simultaneous entries

```c
#define successor_finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
   if (n')
      init_finger_table(n');
      update_others();
      // move keys in (predecessor, n) from successor
   else // n is the only node in the network
      for i = 1 to m
         finger[i].node = n;
         predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
   finger[1].node = n'.find_successor(finger[1].start);
   predecessor = successor.predecessor;
   successor.predecessor = n;
   for i = 1 to m - 1
      if (finger[i + 1].start ∈ [n, finger[i].node))
         finger[i + 1].node = finger[i].node;
      else
         finger[i + 1].node = n'.find_successor(finger[i + 1].start);

// update all nodes whose finger
// tables should refer to n
n.update_others()
   for i = 1 to m
      // find last node p whose i\textsuperscript{th} finger might be n
      p = find_predecessor(n - 2\textsuperscript{i-1});
      p.update_finger_table(n, i);

// if s is i\textsuperscript{th} finger of n, update n's finger table with s
n.update_finger_table(s, i)
   if (s ∈ [n, finger[i].node])
      finger[i].node = s;
   p = predecessor; // get first node preceding n
   p.update_finger_table(s, i);
```
Simultaneous entries

\( n.\text{join}(n') \)

\[
\begin{align*}
\text{predecessor} & = \text{nil}; \\
\text{successor} & = n'.\text{find_successor}(n); \\
\end{align*}
\]

// periodically verify n’s immediate successor; 
// and tell the successor about n.
\( n.\text{stabilize}() \)

\[
\begin{align*}
x & = \text{successor.predecessor}; \\
\text{if} \ (x \in (n, \text{successor})) \\
\text{successor} & = x; \\
\text{successor.notify}(n); \\
\end{align*}
\]

// n' thinks it might be our predecessor.
\( n.\text{notify}(n') \)

\[
\begin{align*}
\text{if} \ (\text{predecessor is nil or n'} \in (\text{predecessor}, n)) \\
\text{predecessor} & = n'; \\
\end{align*}
\]

// periodically refresh finger table entries.
\( n.\text{fix_fingers}() \)

\[
\begin{align*}
i & = \text{random index} > 1 \text{ into } \text{finger}[]; \\
\text{finger}[^i].\text{node} & = \text{find_successor(\text{finger}[^i].\text{start})}; \\
\end{align*}
\]

- Simultaneous entries may cause:
  - Incorrect finger tables: performance degradation
  - Incorrect successors: lookup failure

- New entries degrade the efficiency of the fingers when they place themselves in between the finger.start and the node, forcing the use of successors
The requirement for correct operation is the successor.

Each node maintains a list of $O(\log N)$ successors to be able to recover from failures. If the successor fails, the next one is used.

This allows a Chord network to, with high probability, survive the simultaneous departure of $N/2$ nodes.

If a node is forwarding a message, it will try the previous fingers and afterwards the list of successors.

Successor list may be used for data replication.
Any question?