Object Location

• Problem: Find a close copy of an object in a large network

• Solution should:
  – Find object if it exists
  – Find a close copy of the object (no round trip to Siberia to find an object next door)
  – Balance the load
  – Handle changing participant set
Tapestry: Motivation and Algorithms
Two Simple Solutions

• Solution 1: One central directory
  – Must travel far even for nearby objects
  – Load not balanced

• Solution 2: Every node has a directory.
  – Very expensive to add an object to system
  – (but has locality!)
Better Solution: Hierarchy

• Have many levels of directories
  – Check lowest, then second lowest, and so on until the highest.
  – Many low-level directories, one highest level
  – If object is nearby, it is found quickly

• Problems
  – How to build hierarchy?
    Assume info given?
  – How to load balance?
Random Hierarchy

• At random
  – 1 in 16 nodes level-1 directories.
  – 1 in 256 level-2 directories
  – …
  – Closest level-1 node is your “local” directory

• Load Balance
  – Create 16 types of level-1 directories
  – Create 256 types of level-2 directories
  – …
Tapestry! [PRR]

• Nodes are directories. Node with ID 1234 is a 1 directory, 12 directory, 123 directory….

• Object 5678 is found in closest 5 directory, closest 56 directory, …

• (Cannot do quite this—tables would be too large.)
## How Does Tapestry Fit?

<table>
<thead>
<tr>
<th>System</th>
<th>Insertion</th>
<th>Find</th>
<th>Locality</th>
<th>Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central Directory</td>
<td>O(1)</td>
<td>O(1)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No directory</td>
<td>&gt; O(n)</td>
<td>O(1)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CAN</td>
<td>O(r)</td>
<td>O(rn^{1/r})</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Chord</td>
<td>O(log^2 n)</td>
<td>O(log n)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Pastry</td>
<td>O(log^2 n)</td>
<td>O(log n)</td>
<td>Maybe</td>
<td>Yes</td>
</tr>
<tr>
<td>Tapestry</td>
<td>O(log^2 n)</td>
<td>O(log n)</td>
<td>Some</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Insertion

• Find node with closest matching ID (surrogate) and get preliminary neighbor table
• Find all nodes that need to route to new node via multicast
• Optimize neighbor table
Acknowledged Multicast Algorithm

Locates & Contacts all nodes with a given prefix

- Create a tree based on IDs as we go
- Starting node knows when all nodes reached
- Nodes send acks when all children reached

The node then sends to any
?0345, any ?1345, any
?3345, etc. if possible

```
5431?
  ?4345 sends to
  04345, 54345... if
  they exist

5434?
  54340
  54345
```
Optimize Neighbor Table

• Idea:
  – Given level-\(i\) list, find all level-\((i-1)\) pointing to level-\(i\) list.
  – Trim list
  – Repeat

• Level-\((i-1)\) node in little circle must must point to a level-\(I\) node in the big circle
Simultaneous Insertions

- Two nodes *conflict* if there is no ordering the network can agree on.
  - A arrives *before* B one place, but *after* B some other place
- Modify multicast algorithm so two conflicting nodes find each other.
  - Send down hole being filled (same hole)
  - Send down "watch list" of prefixes (diff hole)
- Nodes getting multicast
  - If node on watchlist found, forward information
  - If hole already filled, send to all such nodes
Locking Pointers

• Multicast assumes that chosen node can forward message.
• Inserting nodes have incomplete information.

So…
• Pointers are added as “locked”. When multicast for that node returns, pointers are unlocked.
• Multicasts are sent to one unlocked pointer and all locked pointers.
• Locked pointers may not be deleted.