This lecture is based on material from the following papers, all available on the course web site:
“Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing”
“Distributed Object Location in a Dynamic Network”
“Tapestry: A Resilient Global-Scale Overlay for Service Deployment”
Overlay Routing Concerns

- **Stretch**
  - routing delay penalty (RDP)
- **Load balancing**
  - popular object located on only one node
What to Do?

- Have multiple copies of objects at well distributed nodes
  - How many?
  - If you have an object at all nodes, what is the cost of read, what is the cost of insert/delete?
  - What if you have one object?
- Take communication distance into account when setting up overlay networks
Tapestry

- Assign each block a unique m-bit ID
  - crypto hash of its contents
- Assign each computer a unique m-bit ID
- Store multiple copies of blocks each at a number of computers
- Store block addresses at computer that has closest ID
  - addresses are cached at other nodes
- Route requests for that block to that computer
  - request is redirected to nearest computer that has copy of block
How to Route?

3312  2302  0331
1332  2130  3311
3111  1331  1001
0121  3320  3120
3122
Notice that 3312 can be a member of its own routing table.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx</td>
<td>0331</td>
<td>1332</td>
<td>2302</td>
<td>3312</td>
</tr>
<tr>
<td></td>
<td>128.148.128.173</td>
<td>128.138.117.92</td>
<td>128.118.165.27</td>
<td>128.213.97.6</td>
</tr>
<tr>
<td>3xx</td>
<td>-</td>
<td>3111</td>
<td>-</td>
<td>3312</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128.172.53.237</td>
<td></td>
<td>128.213.97.6</td>
</tr>
<tr>
<td>33x</td>
<td>-</td>
<td>3311</td>
<td>3320</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128.12.236.81</td>
<td>128.248.192.76</td>
<td></td>
</tr>
<tr>
<td>331</td>
<td>-</td>
<td>3311</td>
<td>3312</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128.12.236.81</td>
<td>128.213.97.6</td>
<td></td>
</tr>
</tbody>
</table>
The routing algorithm is as shown in the slide. Here base is the base of the number system used to form the digits: if base is 16, then we treat the node hashes as being in hexadecimal and thus the routing tables have 16 columns. d is the number of base base digits in the hash, which is equal to ceiling(log_{base}(2^m)), where m is the number of bits in the hash. step is which step we are in the path from the source to the destination, starting at 1. There are d steps from the source to the destination.
Surrogate Routing

- Store object’s location list at unique computer whose hash is “closest” to the object’s hash
  - unique computer known as the root
  - all routes to object’s hash reach the root regardless of the starting point
  - the path to this root goes through various “surrogate” nodes
    - if there is a hole in the neighbor table corresponding to the “next digit”, then choose the first non-empty entry in the row that’s greater (mod base) than the one desired
  - the node routed to is the surrogate
How?

- If no next hop exists, try the next larger digit, mod base
  - each neighbor-table row must have at least one entry
    - why?
  - if any two neighbor-table rows (of different nodes) share the same prefix, they must agree on which entries are null
    - why?
### Neighbor Tables for 3312 and 3320

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>0331</td>
<td>1332</td>
<td>2302</td>
<td>3312</td>
</tr>
<tr>
<td></td>
<td>128.134.0.0</td>
<td>128.134.1.0</td>
<td>128.134.10.0</td>
<td>128.134.10.255</td>
</tr>
<tr>
<td>3xxx</td>
<td>--</td>
<td>3111</td>
<td>--</td>
<td>3312</td>
</tr>
<tr>
<td></td>
<td>128.134.0.0</td>
<td>128.134.254.255</td>
<td>--</td>
<td>128.134.254.255</td>
</tr>
<tr>
<td>33xx</td>
<td>--</td>
<td>3312</td>
<td>3320</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>128.134.0.0</td>
<td>128.134.10.0</td>
<td>128.134.10.255</td>
<td>--</td>
</tr>
<tr>
<td>331x</td>
<td>--</td>
<td>3311</td>
<td>3312</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>128.134.0.0</td>
<td>128.134.10.0</td>
<td>128.134.10.255</td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxx</td>
<td>0121</td>
<td>1001</td>
<td>2130</td>
<td>3311</td>
</tr>
<tr>
<td></td>
<td>128.197.168.0</td>
<td>128.197.254.0</td>
<td>128.197.254.0</td>
<td>128.197.254.0</td>
</tr>
<tr>
<td>3xxx</td>
<td>--</td>
<td>3120</td>
<td>--</td>
<td>3320</td>
</tr>
<tr>
<td></td>
<td>128.197.168.0</td>
<td>128.197.254.0</td>
<td>--</td>
<td>128.197.254.0</td>
</tr>
<tr>
<td>33xx</td>
<td>--</td>
<td>3311</td>
<td>3320</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>128.197.168.0</td>
<td>128.197.254.0</td>
<td>128.197.254.0</td>
<td>--</td>
</tr>
<tr>
<td>332x</td>
<td>3320</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>128.197.168.0</td>
<td>128.197.254.0</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
Surrogate Routing Algorithm

// executed at each node in route to destination
NextHop(targetHash, step) {
    nextDigit = digit(targetHash, step)
    while ((next = table[step, nextDigit]) == NULL)
        nextDigit += 1 mod base
    return next
}

digit(num, pos) {
    return ((num/base^{d-pos})%base)
}
Surrogate Routing for 3021
Performance and Redundancy

- For any particular neighbor-table entry, there may be a number of possible valid next hops
  - all of them work
  - choose the one that’s “closest”
    - communication delay makes sense for this
  - if a next hop can’t be reached
    - use one of the other possible next hops
    - store some number of them in table
  - “secondary entries”

Tapestry stores two additional “next hops” for each neighbor-table entry
Multiple nodes may have a copy of an object. They each “publish” this fact by sending a message to the object’s root, which records the fact that the sender has a copy of the object. Each node in the path to the root caches the fact that the sender has a copy of the object, so that queriers (such as node D in the slide), who send a message to the root to find out where copies of the object reside, are likely to find this information cached at a node that’s on their path to the root, thus making unnecessary having to go all the way to the root.
If the root node fails, then all its information is lost. However, the object-location information still resides in the caches.
Soft State

- State information times out
  - e.g., reference to node holding an object
- Must be periodically reestablished
  - nodes must periodically republish their objects
Redundant Redundancy

- When “root nodes” of objects disappear, it may take some time before re-publication is effective
  - solution: extra root nodes
  - how?
    - “salt” the hashes
      - append a small integer before hashing
      - multiple hashes for one object: multiple roots for the object
The procedure for adding a node we give here isn’t the best approach, but it will help us better understand how things work. After we look at this procedure we’ll sketch some improvements.
Node G (the “gateway” node, 3312) is used to fill in the first row of n’s neighbor table. At the moment we’re not concerned about optimizing communication speed by linking nodes to their nearest neighbors; our concern is strictly connectivity.
Using prefix routing, the next hop in the route to n’s root is also through node 3312. So we copy the second row of its neighbor table into n’s table. Note that this row doesn’t have an entry for the next digit of n’s prefix. Thus the first surrogate node would be 3111. However, since we’re constructing n’s neighbor table, we fill the hole with n’s hash (3001).
Since we had to switch to a surrogate on the way to n’s root, the next row of n’s neighbor table contains just the self-entry for n itself. On the right, we continue to search for n’s root.
Again, we fill in the next (and last) row of n’s table with a self entry. On the right, we establish that 3111 was n’s root.
Notifying Others

- Need to insert $n$ in all neighbor-table entries that are empty where $n$ should go
- Before $n$ was added, any search for $n$ ended up at its root
- Proceed backwards from root
  - each neighbor table includes back pointers to all nodes that route to it
  - *flooding* procedure:
    - on receipt of notification
      - if routing table contains hole where $n$ goes
        - Insert $n$
        - notify neighbors via back pointers
### A Problem

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx</td>
<td>0121 128.148.158.13</td>
<td>1001 128.16.11.192</td>
<td>2130 128.113.226.127</td>
<td>3311 128.12.236.81</td>
</tr>
<tr>
<td>3xx</td>
<td></td>
<td>3120 128.152.253.247</td>
<td></td>
<td>3320 128.248.192.75</td>
</tr>
<tr>
<td>33x</td>
<td></td>
<td>3311 128.12.258.81</td>
<td>3320 128.248.192.75</td>
<td></td>
</tr>
<tr>
<td>332x</td>
<td>3320 128.248.192.75</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Node 3322 is added
- Object 3321’s surrogate was 3320
  - now it's 3322
  - what about all the location info that was stored assuming 3320?
Doing a Better Job …

- Find all nodes for which new node fills holes in neighbor tables
  - propagate new node on spanning tree of just the relevant nodes
- Handle “re-rooted” objects
  - efficiently …
- Build new neighbor tables
  - optimizing for closeness
Observation

- Let α be longest common prefix of new node and its root
- All nodes whose neighbor tables contain holes to be filled by new node have this prefix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx</td>
<td>0331</td>
<td>1332</td>
<td>2302</td>
<td>3312</td>
</tr>
<tr>
<td>3xxx</td>
<td>3001</td>
<td>3111</td>
<td>3312</td>
<td></td>
</tr>
<tr>
<td>30xx</td>
<td>3001</td>
<td>3001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>300x</td>
<td>3001</td>
<td>3001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

n’s (3001’s) Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxx</td>
<td>0331</td>
<td>1332</td>
<td>2302</td>
<td>3312</td>
</tr>
<tr>
<td>3xxx</td>
<td>3111</td>
<td>3111</td>
<td>3120</td>
<td></td>
</tr>
<tr>
<td>31xx</td>
<td>3111</td>
<td>3111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>311x</td>
<td>3111</td>
<td>3111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

root’s (3111’s) Table

CS 138
III–30

Copyright © 2015 Thomas W. Doegeper and Rodrigo Fenseca. All rights reserved.
Application

- Send new node’s info to all nodes with prefix α
  - hash and IP address
- How?
  - via spanning tree that reaches all such nodes
  - use “acknowledged multicast”
This code is taken from the paper “Distributed Object Location in a Dynamic Network” and certainly requires some explanation. It is first called at the surrogate node with the longest prefix shared with the new node. The for loop loops over all the digits in the base. We are looking for one neighbor in the neighbor table for each of the prefixes that start with $\alpha$ and contain one more digit. For each such neighbor, the code is called recursively. If there is only one node with the given prefix, then clearly we can’t recurse further. However, eventually all nodes with the given prefix will be contacted and the function supplied as an argument will be applied on each of them.

A leap of faith in syntax (and semantics) comes to play with the recursive calls to `acknowledgedMulticast`. These are implicitly assumed not to be function calls but reliable transmissions of request messages. The variable $S$ acts both as a collector of results and as a synchronization variable. As a collector of results, it is a vector containing the results of applying the function at each of the nodes. As a synchronization variable it is used in the wait statement to cause the caller to wait until all instances of `acknowledgedMulticast` made by the current instance have terminated.

```
Acknowledged Multicast

n.acknowledgedMulticast(\alpha, function) {
    if (notOnlyNodeWithPrefix(\alpha))
        for i = 0 to b-1
            neighbor = neighborWithPrefix(\alpha; i)
            if neighbor exists
                S = neighbor.acknowledgedMulticast(\alpha; i, function)
            else
                apply function
                wait S
                SendAcknowledgement()
}  
```
Using Acknowledged Multicast

• To fill holes in neighbor tables with new node
  – supply function that does this
• To “re-root” object references (move them to the new node)
  – supply function that does this
• To get list of all nodes with given prefix
  – supply function that returns node IDs
Races

- Suppose a search for object x occurs while new node N is being added
  - N becomes the new root for x
- Potential problems:
  1) search arrives at N before object references are transferred to N
  2) search arrives at old root after object references are transferred to N
Problem 1 Solution

- Mark N as “being inserted”
  - on “object not found”
    - forward request to original root
Problem 2 Solution

- Include path with search request
  - on “object not found”
    - check neighbor table to see if path would still be taken
    - i.e., has hole subsequently been filled (by new root)?
    - if so, reroute to new root
Optimizing the Neighbor Table

- Want primary node for each neighbor-table entry to be the one that’s closest
  - secondary nodes should be the next closest
- How can this be constructed for a new node?
Sketch

- Use acknowledged multicast to find all nodes with prefix $\alpha$ (longest prefix in common with root)
- From this set of nodes, construct table row $|\alpha|$
  - determine which are closest to the new node
- From this set of nodes, find all nodes with prefix one shorter than $\alpha$
  - construct table row $|\alpha|-1$, using closest nodes
- etc.