Frequently asked questions from the previous class survey

- Pastry
  - How do you determine the column, after you have found the matching row?
  - Leaf node changes ..
  - Locality, geography, and how variations in the routing table make the routing robust
Topics covered in this lecture

- Pastry (Wrap-up)
- Tapestry
- Chord
  - Mapping of data items
  - Data lookups
  - Finger Table construction
  - Assimilation of peers

Pastry Wrap-up
Repairing leaf sets

- Node that discovers the failure
  - Looks for a live node close to the failed node, and requests copy of that node’s leaf set, \( L' \)
  - This should contain GUIDs that partly overlap those in the node that discovered failure
    - Include one that should replace the failed node
- Other neighboring nodes are informed
  - They perform a similar procedure

Locality

- Pastry routing structure is redundant
  - Multiple routes between pairs of nodes
- Construction of routing tables tries to take advantage of this redundancy
  - Reduce message transmission times by exploiting locality properties of underlying network
Routing table: Exploiting locality. [1/2]

- In the routing table, each row contains 16 entries
  - Entries in the $i^{th}$ row give addresses of 16 nodes with GUIDs with $i-1$ initial hexadecimal digits
  - $i^{th}$ digit takes each of the possible hexadecimal values
- Well-populated Pastry system contains more nodes than can be contained in an individual routing table

Routing table: Exploiting locality. [2/2]

- When routing table is constructed choice is made for each position
  - Between multiple candidates
  - Based on proximity neighbor selection
- Locality metric
  - IP hops or measured latency
Performance of exploiting locality

- Since the information in the routing table is not comprehensive
  - Mechanism does not produce globally optimal routing

- Simulations show that
  - On average, the routing is 30-50% longer than the optimum

Coping with malicious nodes

- Small degree of *randomness* is introduced into route selection
- Randomized to yield a common prefix that is less than the maximum length
  - With a certain probability
- Routes are taken from an earlier row
  - Less optimal, but different than standard version
  - Client transmission succeed in the presence of small numbers of malicious nodes
Tapestry

- Routes messages to nodes based on GUIDs associated with the resources
  - Uses prefix routing in a manner similar to Pastry

- **160-bit** identifiers are used
  - To refer to both objects and nodes that perform routing actions

- For any resource with GUID \( G \), there is a unique root node, with GUID \( R_G \)
  - \( R_G \) is numerically closest to \( G \)
Tapestry Routing [Summary]

- Uses local routing tables, which they also call **neighbor maps**, to route messages
- Routing is digit-by-digit
  - $4^\ast\ast\ast \rightarrow 42^\ast\ast \rightarrow 42A^* \rightarrow 42AD$
- This longest prefix routing is also used by classless interdomain routing (CIDR)

Tapestry: Routing messages

- Each node maintains a **routing table**
  - Entries include node IDs and IP addresses
- This routing table has **multiple levels**
  - Each level contains links to nodes matching a prefix up to a digit position in the ID
  - The $i^{th}$ entry in the $j^{th}$ level at node $N$?
    - Location of the closest node which begins with the $\text{prefix}(N, j-1) + i$
    - E.g. 9th entry of the 4th level for node 325AE is $3259$
Tapestry Routing

- The router for the $n^{th}$ hop
  - Shares a prefix of length $\geq n$ with the destination ID
  - Looks in its $(n+1)^{th}$ level map for entry matching the next digit in the destination ID

- Guarantees that any node in the system can be reached in at most $\log N$ logical hops
  - $N$ is the size of the ID space i.e. $N = 2^{160}$

When a digit cannot be matched?

- Looks for a “close” digit in the routing table
- This approach is called surrogate routing
  - Results in mapping every identifier $G$ to a unique root node $G_R$
Managing a dynamic environment

- Route reliably even when intermediate links are changing or faulty
- Exploit network **path diversity**
  - Via *redundant* routing paths
- Primary links are augmented by **backup-links**
  - Each sharing the same prefix

Managing multiple copies of the resource

- Hosts \( H \) holding replicas of \( G \) periodically invoke \( publish(G) \)
  - Ensures that newly arrived hosts become aware of the existence of \( G \)
- On each invocation of \( publish(G) \)
  - Message is routed from invoker towards node \( R_G \)
  - On receipt of a publish message \( R_G \) enters \( (G, IP_H) \)
    - The mapping between \( G \) and IP address of \( H \)
    - Each node in the publication path caches the same mapping
Managing multiple copies of the resource

- When nodes hold multiple (G, IP) mappings for the same GUID?
  - They are sorted by network distance to the IP address
- Results in selection of nearest available replica of the object

An example of managing replicas using Tapestry

- Replica retrieval path
- Publish Path

September 17, 2019
Professor: SHRIDEEP PALLICKARA
The Chord System

- Assigns IDs to keys and nodes from the same 1-dimensional ID space
- Nodes are organized into a **ring**
- Data item with key \( k \) is mapped to a node with the **smallest** \( \text{id} \geq k \)
  - Also referred to as \( \text{successor}(k) \)
Mapping of data items to nodes in Chord

- **Actual Node**: 0, 1, 2, 3, 4, 5, 6, 7
- **Associated data keys**:
  - Node 1: {0, 1}
  - Node 2: {2, 3, 4}
  - Node 3: {5, 6, 7}
  - Node 14: {13, 14, 15}
  - Node 12: {8, 9, 10, 11, 12}
  - Node 11: {8, 9, 10, 11, 12}
  - Node 10: {8, 9, 10, 11, 12}
  - Node 9: {8, 9, 10, 11, 12}
  - Node 7: {5, 6, 7}
  - Node 6: {5, 6, 7}
  - Node 5: {5, 6, 7}

Chord lookups

- **N** is the number of possible nodes in the system
- Each node maintains a **finger table**
  - With \( \log N \) entries
  - Entries contain IP addresses of nodes
  - Half-way around the ID space from it
  - \( 1/4^n, 1/8^n, \ldots \) in powers of two
  - Ensures node can forward lookup query to at least \( 1/2 \) of the remaining ID-space distance to key
  - Lookups in \( O(\log N) \)
Storing keys and forwarding lookups

- An entity with key $k$ falls under the *jurisdiction* of node with the *smallest identifier* $id$
  - $id \geq k$
  - Referred to as the successor of $k$ or $\text{succ}(k)$

- A node *forwards* query for key $k$ to node (in its FT) with highest ID $\leq k$
  - The exception is ONLY when the first entry is greater than $k$

Chord lookup example for $k=54$
When a node wants to join

- Generate a random id
  - Probability of collisions is low
- \texttt{lookup(id)}
  - Will return \texttt{successor(id)}
- Contact \texttt{successor(id)} and its predecessor
  - Insert self in the ring
  - \textbf{Transfer} data items
    - All keys must be fetched from the new node's successor

An example of inserting a new node

- \texttt{Succ(12) = 15}
- \texttt{Pred(12) = 7}
- \text{Associated data keys: } \{8, 9, 10, 11, 12\}
- \text{New node 10 will be inserted}
- \text{Actual Node}
An example of inserting a new node

Finger Table in Chord

- Chord uses an $m$-bit identifier space
  - $2^m$ possible peers
- Each node, $p$, in Chord maintains a Finger Table with $m$-entries
  - $FT_p[i] = \text{succ}(p + 2^{i-1})$
We are looking at a 5-bit ID space. IDs go from 0 through \((2^5 - 1)\).

### Constructing the Finger Table: Node 1

\[
\text{succ}(k) = \text{Smallest id } \geq k
\]

<table>
<thead>
<tr>
<th>Index</th>
<th>succ((p + 2^{i-1}))</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>succ(1 + 1)</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>succ(1 + 2)</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>succ(1 + 4)</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>succ(1 + 8)</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>succ(1 + 16)</td>
<td>18</td>
</tr>
</tbody>
</table>

### Constructing the Finger Table: Node 4

\[
\text{succ}(k) = \text{Smallest id } \geq k
\]

<table>
<thead>
<tr>
<th>Index</th>
<th>succ((p + 2^{i-1}))</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>succ(4 + 1)</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>succ(4 + 2)</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>succ(4 + 4)</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>succ(4 + 8)</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>succ(4 + 16)</td>
<td>20</td>
</tr>
</tbody>
</table>
Constructing the Finger Table: Node 9

\[ \text{succ}(k) = \text{Smallest id } \geq k \]

<table>
<thead>
<tr>
<th>Index</th>
<th>succ((p + 2^{i-1}))</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>succ(9 + 1)</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>succ(9 + 2)</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>succ(9 + 4)</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>succ(9 + 8)</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>succ(9 + 16)</td>
<td>28</td>
</tr>
</tbody>
</table>

\[ \text{if (val } \geq 2^m) \{ \text{val = val (mod } 2^m) \} \]

Constructing the Finger Table: Node 28

\[ \text{succ}(k) = \text{Smallest id } \geq k \]

<table>
<thead>
<tr>
<th>Index</th>
<th>succ((p + 2^{i-1}))</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>succ(28 + 1)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>succ(28 + 2)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>succ(28 + 4)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>succ(28 + 8)</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>succ(28 + 16)</td>
<td>14</td>
</tr>
</tbody>
</table>
Using the finger table to route queries:
Make sure you don’t overshoot

- To lookup a key $k$, node $p$ will forward query to node $q$ with index $j$ in $p$’s FT where:
  
  - Node with greatest ID less than or equal to $k$
  
  \[
  q = FT_p[j] \leq k < FT_p[j+1]
  \]
  
  OR
  
  \[
  q = FT_p[1] \text{ when } p < k < FT_p[1]
  \]

  First entry ONLY if its ID is greater than $k$

Stop forwarding the query when you are the target node

- A node is **responsible** for keys that fall in the range
  
  - $key > \text{predecessor}$
  
  - $key \leq \text{self}$
\[ q = FT_p[j] \leq k < FT_p[j+1] \]
\[ q = FT_p[1] \text{ when } p < k < FT_p[1] \]
Keeping the finger table up-to-date:
At node $q$, $FT_q[1]$ must be accurate

1. **Contact** $succ(q+1)$ \{This is $FT_q[1]$\}
   - Have it return its predecessor

2. If $q = pred(\ succ(q+1) \ )$
   - Everything is fine

3. Otherwise:
   - There is a new node $p$ such that $q < p \leq succ(q+1)$
   - $FT_q[1] = p$
   - Check if $p$ has recorded $q$ as its predecessor
     - No? Go to step (1)

AN EXAMPLE OF NODES JOINING IN CHORD
An example of inserting a new node N-4:
Node-4 comes in and contacts Node-1

Installing successor at Node-1
Updating the FT at N-1

An example of inserting a new node N-7:
N-7 contacts N-1 for filling its FT
N-7 informs N-1 that it (N-7) is now N-1’s predecessor

Since 7 is closer it is installed as the predecessor of 1

When N-1 updates its FT later on ...

Succ(7) = 1
Pred(1) = 7
Succ(1) = 4

Succ(4) = 1
Pred(4) = 1
N-4 contacts N-1 to see if it is still its predecessor ... and installs N-7 as its successor

When the FT at N-4 is updated ...
The contents of this slide-set are based on the following references
