Hashing Nodes and Data

- Hashing is random, can expect nodes to be evenly distributed in key space
- Nodes hash IP addresses to key space
- Store data in the successor of the data item's key
- Property: If each node maintains successor, can find any data item...

SUCCESSOR(6)=0
SUCCESSOR(2)=3
SUCCESSOR(1)=1

Nodes have a successor pointer
- Each node maintains successors.
- PROPERTY: Each data item stores in the successor of the node in the key space.
- Property: Nodes hash IP addresses to key space.
- Property: If each node maintains successors, can find any data item in the key space.

Chord

Compare Search in Several Peer-to-Peer Systems

- easy if static, but must deal with node arrivals and departures
- uses consistent hashing
- emphasis on good cryptographic performance
- map key to value
- A structured peer-to-peer system

Chord

http://merlot.usc.edu/cs551-f12
at each step, we halve the remaining distance (in key space) to the target

Challenge: maintaining finger tables!

Finger tables enable logarithmic lookup

Ex: look for key y

case 1: y is just beyond x+2

forward to successor(x+2)

way more than half the distance to y

case 2: y is just inside x+2

forward to successor(x+2)

a little over half the distance to y

case 3: y is just beyond x+2

forward to successor(x+2)

way more than half the distance to y

Finger tables enable logarithmic lookup

Improving Search Performance with Finger Tables (Cont...)
Node Joins

- Can always fall back on them to find a key.
- Must keep successors and finger table current.
- Use successors for correctness; must update it, but can tolerate temporary errors.
- Use finger table for performance.
- Keep successor and predecessor so we can update our neighbors.

Key observation: can find successors and fingers by doing a lookup on the existing Chord ring!

### Finding Predecessor and Successor

**find_successor(key)**

```java
n = find_predecessor(key);
return n.successor;
```

**find_predecessor(key)**

```java
n = node;
while (key ∉ (n,n.successor])
  n = n.closest_preceding_finger(key);
return n;
```

**closest_preceding_finger(key)**

```java
for (i=m; i > 0; i--)
  if (finger[i].node ∈ (node,key))
    return finger[i].node;
return node;
```

### Join Example

When a new node enters, it establishes its successor and predecessor and then builds its finger table, and moves any keys it now "owns".

#### Before Node 6 Joins

<table>
<thead>
<tr>
<th>Node</th>
<th>Successor</th>
<th>Finger Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>[1,2)</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>[2,4)</td>
</tr>
</tbody>
</table>

#### After Node 6 Joins

<table>
<thead>
<tr>
<th>Node</th>
<th>Successor</th>
<th>Finger Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>[1,2)</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>[1,2) [2,4)</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>[2,4) [2,4)</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>[0,2) [2,4)</td>
</tr>
</tbody>
</table>

### Robustness

- Can use these to replicate data.
- Keep access to list of successors.
- Balancing with unbalanced elifs:
  - For some churn, network sizes of n and 2n.
  - More frequent observations: networks can be inconsistent.
  - Reduce frequent updates this way.
  - Also, pick and verify a random neighbor entry.
  - Access keys like predecessors if necessary.
  - Else, your own access based on this.
- Fix your own successor if necessary.
  - Every node selects successor off predecessor.
  - Coordinator reboots in case of incorrect.

### Applications

- User keys if not "own" predecessors and then builds the finger table and moves.

### Chord Performance

- How long does it take to get to the node that stores a key?
- Performance dominated by lookup cost.
- Chord promises few O(logN) hops on the overlay.
- This is often the problem with overlay networks.

<table>
<thead>
<tr>
<th>Node</th>
<th>Successor</th>
<th>Finger Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>[1,2)</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>[1,2) [2,4)</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>[2,4) [2,4)</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>[0,2) [2,4)</td>
</tr>
</tbody>
</table>

### Applications

- Multicast and Anycast (using rendezvous)

<table>
<thead>
<tr>
<th>File Systems</th>
<th>Modelling and Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicast</td>
<td>Anycast</td>
</tr>
<tr>
<td>(id,R)</td>
<td>(id,R)</td>
</tr>
<tr>
<td>Receiver</td>
<td>Receiver</td>
</tr>
<tr>
<td>Sender</td>
<td>Sender</td>
</tr>
</tbody>
</table>

### Node Joins

- Keep successors and predecessors so we can update our configuration.
- Keep successor and predecessor so we can update our configuration.
- Can make updates if break on it to find a key.
- Keep successors andfinger tables current.
- Must keep successors andfinger tables current.