Distributed Hash Tables (DHT)

Hash table

<table>
<thead>
<tr>
<th>k₀</th>
<th>v₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>k₁</td>
<td>v₁</td>
</tr>
<tr>
<td>k₂</td>
<td>v₂</td>
</tr>
<tr>
<td>k₃</td>
<td>v₃</td>
</tr>
<tr>
<td>k₄</td>
<td>v₄</td>
</tr>
<tr>
<td>k₅</td>
<td>v₅</td>
</tr>
</tbody>
</table>
Chord - Overview

Hash table

<table>
<thead>
<tr>
<th>k₀</th>
<th>v₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>k₁</td>
<td>v₁</td>
</tr>
<tr>
<td>k₂</td>
<td>v₂</td>
</tr>
<tr>
<td>k₃</td>
<td>v₃</td>
</tr>
<tr>
<td>k₄</td>
<td>v₄</td>
</tr>
<tr>
<td>k₅</td>
<td>v₅</td>
</tr>
</tbody>
</table>
Today

• Chord

• Tackling DHT Issues
  • Replication/Fault tolerance
  • Popular items

• Tapestry
  • Routing
  • Adding Nodes
Chord - Overview

Identifier ring over hash space $2^m$

$2^{m-1}$

$0$ $1$

Each node maintains:
- Finger table
- Success
- Predecessor

Finger table for node at id $i$

<table>
<thead>
<tr>
<th>finger</th>
<th>node id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>succ($i$)</td>
</tr>
<tr>
<td>2</td>
<td>succ($i + 2$)</td>
</tr>
<tr>
<td>$j$</td>
<td>succ($i + 2^{j-1}$)</td>
</tr>
</tbody>
</table>

node id = hash( node )
key id = hash( key )
Chord - Overview

Identifier ring

0 = node

11 = key
Chord - Overview

Identifier ring

0 = node
= key

<table>
<thead>
<tr>
<th>finger</th>
<th>node id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>succ(i)</td>
</tr>
<tr>
<td>2</td>
<td>succ(i + 2)</td>
</tr>
<tr>
<td>3</td>
<td>succ(i + 2²)</td>
</tr>
<tr>
<td>4</td>
<td>succ(i + 2³)</td>
</tr>
</tbody>
</table>
Chord - Overview

Identifier ring

= node

= key

<table>
<thead>
<tr>
<th>finger</th>
<th>node id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>succ(4)</td>
</tr>
<tr>
<td>2</td>
<td>succ(4 + 2)</td>
</tr>
<tr>
<td>3</td>
<td>succ(4 + 2^2)</td>
</tr>
<tr>
<td>4</td>
<td>succ(4 + 2^3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node id</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Chord - Lookup

Identifier ring

```
<table>
<thead>
<tr>
<th>finger</th>
<th>node id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
lookup(10)
follow finger 3 to node id 8
node id 8 identifies as predecessor of id 10
```

```
join():
    self.predecessor = null
    self.successor = find_successor(self)

stabilize():
    p = self.successor.predecessor
    if p between (self, self.successor):
        self.successor = p
        self.successor.notify(self)

notify(n):
    if self.predecessor == null ||
        n between (self.predecessor, self):
        self.predecessor = n
```

```
Hops?
Each finger lookup halves distance to key
O(log N)
```
### Join Code

```python
join():
    self.predecessor = null
    self.successor = find_successor(self)
```

### Stabilization Code

```python
stabilize():
p = self.successor.predecessor
if p between (self, self.successor):
    self.successor = p
    self.successor.notify(self)
```

### Notify Code

```python
notify(n):
    if self.predecessor == null ||
        n between (self.predecessor, self):
        self.predecessor = n
```
Chord - Joins + Stabilization

Join:
- `self.predecessor = null`
- `self.successor = find_successor(self)`

Stabilize:
- `p = self.successor.predecessor`
- If `p` is between `(self, self.successor)`:
  - `self.successor = p`
  - `self.successor.notify(self)`

Notify(n):
- If `self.predecessor == null || n` is between `(self.predecessor, self)`:
  - `self.predecessor = n`
Chord - Joins + Stabilization

Identifier ring

0
predecessor = 6
successor = 11

1
predecessor = 6
successor = 11

2
predecessor = 5
successor = 8

4
predecessor = 4
successor = 6

8
predecessor = 5
successor = 8

Outcomes of incomplete stabilization:
1. Lookup unaffected
2. Fingers out-dated, successors correct -> lookup slow but correct
3. Successors in lookup region still stabilizing -> lookup fails

join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
    self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
    n between (self.predecessor, self):
    self.predecessor = n
Today

- Chord

- Tackling DHT Issues
  - Replication/Fault tolerance
  - Popular items

- Tapestry
  - Routing
  - Adding Nodes
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
Chord - Failure + Replication

Monitor list of \( k \) successors

Keys replicated on all \( k \) successors
Replication: Challenge!!!!

Copies of an object are identical
Hence, same hash
Replication: Re-salting

- When replicating object, add a little salt.
- Everyone knows the salt

Why does this work?
MD5(000000) = 670b14728a....
MD5(0000001) = da292230f0....
Replication: Multiple DHTS

Create multiple DHTS and add replication to different DHTs. Each DHTS has a different hash algorithm.
Replication: ReSalt V. Multiple DHTs
Today

• Chord

• Tackling DHT Issues
  • Replication/Fault tolerance
  • Popular items

• Tapestry
  • Routing
  • Adding Nodes
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? Using Prefix Lookup

look up: 3122
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”

Route Table for Node: 3312

<table>
<thead>
<tr>
<th>Row</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>—</td>
<td>3111</td>
<td>—</td>
<td>3312</td>
</tr>
<tr>
<td>33XX</td>
<td>—</td>
<td>3311</td>
<td>3320</td>
<td>—</td>
</tr>
<tr>
<td>331X</td>
<td>—</td>
<td>3311</td>
<td>3312</td>
<td>—</td>
</tr>
</tbody>
</table>

Row determines prefix length
Col determines digit
### Route Table for Node: 3312

<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>—</td>
<td>3111</td>
<td>—</td>
<td>3312</td>
</tr>
<tr>
<td>33XX</td>
<td>—</td>
<td>3312</td>
<td>3320</td>
<td>—</td>
</tr>
<tr>
<td>331X</td>
<td>—</td>
<td>3311</td>
<td>3312</td>
<td>—</td>
</tr>
</tbody>
</table>

### Route Table for Node: 3320

<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</td>
<td>1001</td>
<td>2130</td>
<td>3311</td>
</tr>
<tr>
<td>3XXX</td>
<td>—</td>
<td>3120</td>
<td>—</td>
<td>3320</td>
</tr>
<tr>
<td>33XX</td>
<td>—</td>
<td>3311</td>
<td>3320</td>
<td>—</td>
</tr>
<tr>
<td>332X</td>
<td>3320</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

![Diagram](attachment:image_url)
Routing Algorithm

// executed at each node in route to destination
NextHop(targetHash, step) {
    nextDigit = digit(targetHash, step)
    return(table[step, nextDigit])
}

digit(num, pos) {
    return ((num/base^pos)%base)
}

Route Table for Node: 3312

<table>
<thead>
<tr>
<th>XXXX</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>XXXX</td>
<td>03</td>
<td>1332</td>
<td>2302</td>
<td>3312</td>
</tr>
<tr>
<td>3XXX</td>
<td>3111</td>
<td>—</td>
<td>—</td>
<td>3312</td>
</tr>
<tr>
<td>33XX</td>
<td>—</td>
<td>3312</td>
<td>3320</td>
<td>—</td>
</tr>
<tr>
<td>331X</td>
<td>—</td>
<td>3311</td>
<td>3312</td>
<td>—</td>
</tr>
</tbody>
</table>
Routing Algorithm

```plaintext
// executed at each node in route to destination
NextHop(targetHash, step) {
    nextDigit = digit(targetHash, step)
    return(table[step, nextDigit])
}

digit(num, pos) {
    return ((num/base^pos)%base)
}
```

Route Table for Node: 3312

<table>
<thead>
<tr>
<th>XXXX</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0331</td>
<td>1332</td>
<td>2302</td>
<td>3312</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td>3312</td>
<td>3320</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>33XX</td>
<td>—</td>
<td>3312</td>
<td>3320</td>
<td>—</td>
</tr>
<tr>
<td>331X</td>
<td>—</td>
<td>3311</td>
<td>3312</td>
<td>—</td>
</tr>
</tbody>
</table>
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?
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
Publishing

Diagram showing the flow of data through various nodes labeled A, B, C, and D. The diagram illustrates the process of data publishing and handling through different stages.
Failure
Failure
Soft State

• State information times out
  – e.g., reference to node holding an object
• Must be periodically reestablished
  – nodes must periodically republish their objects
Recovery
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