Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications

Ion Stoica, Robert Morris, David Liben-nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan
Introduction

• Functionality:
  given a key, it maps the key onto a node

• Feature
  maintains $O(\log N)$ entries of node information
  resolves lookup via $O(\log N)$ messages
Chord Protocol

- Consistent Hashing
  minimal disruption when nodes enter and leave the network

```
node → m-bit identifier
```

```
key → m-bit identifier
```

```
Identifier circle (chord ring)
size: \(2^m\)
```

```
assign to node = successor(key)
```

```
position on \(2^m\)
```

```
0 \(\rightarrow 2^m-1\)
```

Chord Protocol

- Assign Keys to Nodes

2. Identifier circle (ring) consisting of ten nodes storing five keys.
Chord Protocol

- Simple Key Location

```java
// ask node n to find the successor of id
n.find_successor(id)
    if (id ∈ (n, successor])
        return successor;
    else
        // forward the query around the circle
        return successor.find_successor(id);
```
Chord Protocol

- **Scalable Key Location** – variables for node

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{finger}[k]$</td>
<td>first node on circle that succeeds $(n + 2^{k-1}) \mod 2^m$, $1 \leq k \leq m$</td>
</tr>
<tr>
<td>$\text{successor}$</td>
<td>the next node on the identifier circle; $\text{finger}[1].\text{node}$</td>
</tr>
<tr>
<td>$\text{predecessor}$</td>
<td>the previous node on the identifier circle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>$\text{finger}[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\text{successor}(n+1)$</td>
</tr>
<tr>
<td>2</td>
<td>$\text{successor}(n+2)$</td>
</tr>
<tr>
<td>3</td>
<td>$\text{successor}(n+4)$</td>
</tr>
<tr>
<td>4</td>
<td>$\text{successor}(n+8)$</td>
</tr>
<tr>
<td>5</td>
<td>$\text{successor}(n+16)$</td>
</tr>
<tr>
<td>6</td>
<td>$\text{successor}(n+32)$</td>
</tr>
</tbody>
</table>
Chord Protocol

- Scalable Key Location – key lookup process

```plaintext
// ask node n to find the successor of id
n.find_successor(id)
    if (id ∈ (n, successor])
        return successor;
    else
        n' = closest_preceding_node(id);
        return n'.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
    for i = m downto 1
        if (finger[i] ∈ (n, id))
            return finger[i];
    return n;
```
Chord Protocol

- Node Joins and Stabilization

```python
// create a new Chord ring.
n.create()
    predecessor = nil;
    successor = n;

// join a Chord ring containing node n'.
n.join(n')
    predecessor = nil;
    successor = n'.find_successor(n);

// called periodically. verifies n's immediate
// successor, and tells the successor about n.
n.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
        successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n' ∈ (predecessor, n))
        predecessor = n';

// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
n.fix_fingers()
    next = next + 1;
    if (next > m)
        next = 1;
    finger[next] = find_successor(n + 2^{next−1});

// called periodically. checks whether predecessor has failed.
n.check_predecessor()
    if (predecessor has failed)
        predecessor = nil;
```
Chord Protocol

- Node Joins and Stabilization

N26 joins N32

successor(N21)

N21

K24

K30
Chord Protocol

- Node Joins and Stabilization

```
N26.join(N?)
{
  predecessor = nil
  successor = N?.find_successor(N26) = N32
}
```
Chord Protocol

- Node Joins and Stabilization

N26 copies K24
Chord Protocol

- Node Joins and Stabilization

```plaintext
N21.stable()
{
    x = successor, predecessor = N26;
    if(x∈(N21,N32))
        N21.successor = x = N26;
    N26.notify(N21);
}

N26.notify(N21)
{
    if(predecessor = nil or N21∈(predecessor,N26))
        predecessor = N21;
}

N32 deletes K24
```
Chord Protocol

- **Node Joins and Stabilization**

  each node can forward a query at least halfway along the remaining distance

  with high probability, the number of forwardings is $O(\log N)$

  with high probability, the number of distinct entries in finger table is $O(\log N)$
Chord Protocol

• Node Joins and Lookup

three cases of lookup before stabilization:
1) finger entries involved is good → OK
2) successor pointers are correct, fingers are not → slow
3) successor pointers are incorrect → fail then retry
Chord Protocol

- Node Joins and Lookup

N stable nodes & N new nodes

successors updated correctly
finger tables not yet updated correctly

lookup(N32)

O(log N) hops to reach N21 +
O(log N) hops to reach N32
Chord Protocol

- Failure and Replication

N14, N21, N32 fail

N8.find_successor(30) = 42

correct answer is 38

Maintain a successor list of size r
Chord Protocol

- Voluntary Node Departures
  
  transfer keys to successor

  notify its predecessor and successor
Simulation

- Protocol Simulator

1. iterative vs. recursive lookup process
2. entry refreshed once every k stabilization rounds
3. notify changes of predecessor to old predecessor
4. notification of node failed
5. no consideration about data transfer
Simulation

- Load Balance
  virtual nodes make it more balanced
Simulation

- Path Length
  
  expected path length is $\frac{1}{2} \log N$
Simulation

- Simultaneous Node Failures

### Table II

Path length and number of timeouts experienced by a lookup as function of the fraction of nodes that fail simultaneously. The first and 99th percentiles are in parentheses. Initially, the network has 1000 nodes.

<table>
<thead>
<tr>
<th>Fraction of failed nodes</th>
<th>Mean path length (1st, 99th percentiles)</th>
<th>Mean num. of timeouts (1st, 99th percentiles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.84 (2, 5)</td>
<td>0.0 (0, 0)</td>
</tr>
<tr>
<td>0.1</td>
<td>4.03 (2, 6)</td>
<td>0.60 (0, 2)</td>
</tr>
<tr>
<td>0.2</td>
<td>4.22 (2, 6)</td>
<td>1.17 (0, 3)</td>
</tr>
<tr>
<td>0.3</td>
<td>4.44 (2, 6)</td>
<td>2.02 (0, 5)</td>
</tr>
<tr>
<td>0.4</td>
<td>4.69 (2, 7)</td>
<td>3.23 (0, 8)</td>
</tr>
<tr>
<td>0.5</td>
<td>5.09 (3, 8)</td>
<td>5.10 (0, 11)</td>
</tr>
</tbody>
</table>
Simulation

- Lookups During Stabilization

<table>
<thead>
<tr>
<th>Node join/leave rate (per second/per stab. period)</th>
<th>Mean path length (1st, 99th percentiles)</th>
<th>Mean num. of timeouts (1st, 99th percentiles)</th>
<th>Lookup failures (per 10,000 lookups)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05 / 1.5</td>
<td>3.90 (1, 9)</td>
<td>0.05 (0, 2)</td>
<td>0</td>
</tr>
<tr>
<td>0.10 / 3</td>
<td>3.83 (1, 9)</td>
<td>0.11 (0, 2)</td>
<td>0</td>
</tr>
<tr>
<td>0.15 / 4.5</td>
<td>3.84 (1, 9)</td>
<td>0.16 (0, 2)</td>
<td>2</td>
</tr>
<tr>
<td>0.20 / 6</td>
<td>3.81 (1, 9)</td>
<td>0.23 (0, 3)</td>
<td>5</td>
</tr>
<tr>
<td>0.25 / 7.5</td>
<td>3.83 (1, 9)</td>
<td>0.30 (0, 3)</td>
<td>6</td>
</tr>
<tr>
<td>0.30 / 9</td>
<td>3.91 (1, 9)</td>
<td>0.34 (0, 4)</td>
<td>8</td>
</tr>
<tr>
<td>0.35 / 10.5</td>
<td>3.94 (1, 10)</td>
<td>0.42 (0, 4)</td>
<td>16</td>
</tr>
<tr>
<td>0.40 / 12</td>
<td>4.06 (1, 10)</td>
<td>0.46 (0, 5)</td>
<td>15</td>
</tr>
</tbody>
</table>
Simulation

• Improving Route Latency

  Why? Random generated identifiers

  maintain a set of alternative nodes for each finger: successors of this finger

  select closest node to forward query
Simulation

- Improving Route Latency

<table>
<thead>
<tr>
<th>Number of fingers’ successors (s)</th>
<th>Stretch (10th, 90th percentiles)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iterative</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3-d space</td>
<td>Transit stub</td>
<td>3-d space</td>
</tr>
<tr>
<td>1</td>
<td>7.8 (4.4, 19.8)</td>
<td>7.2 (4.4, 36.0)</td>
<td>4.5 (2.5, 11.5)</td>
</tr>
<tr>
<td>2</td>
<td>7.2 (3.8, 18.0)</td>
<td>7.1 (4.2, 33.6)</td>
<td>3.5 (2.0, 8.7)</td>
</tr>
<tr>
<td>4</td>
<td>6.1 (3.1, 15.3)</td>
<td>6.4 (3.2, 30.6)</td>
<td>2.7 (1.6, 6.4)</td>
</tr>
<tr>
<td>8</td>
<td>4.7 (2.4, 11.8)</td>
<td>4.9 (1.9, 19.0)</td>
<td>2.1 (1.4, 4.7)</td>
</tr>
<tr>
<td>16</td>
<td>3.4 (1.9, 8.4)</td>
<td>2.2 (1.7, 7.4)</td>
<td>1.7 (1.2, 3.5)</td>
</tr>
</tbody>
</table>

TABLE IV
STRETCH OF LOOKUP LATENCY FOR A CHORD SYSTEM WITH $2^{16}$ NODES WHEN THE LOOKUP IS PERFORMED BOTH IN THE ITERATIVE AND RECURSIVE STYLE. TWO NETWORK MODELS ARE CONSIDERED: A 3-D EUCLIDEAN SPACE AND A TRANSIT-STUB NETWORK.
Conclusion

- **Load Balance** → hashed keys and nodes
- **Decentralization** → P2P application
- **Scalability** → lookup takes $O(\log N)$ hops
- **Availability** → update automatically
- **Flexible naming** → no constraints on keys