Chord - A Distributed Hash Table

Yimei Liao
Outline

- Lookup problem in Peer-to-Peer systems and Solutions
- Chord Algorithm
  - Consistent Hashing
  - Scalable Key Location
  - Node joins
  - Stabilization
- Summary
Peer-to-Peer Systems

- Peer-to-Peer System: self-organizing system of equal, autonomous entities (peers)
  - decentralized resource usage
  - decentralized self-organization
  - where to store? where to get?

- Solutions
  - centralized servers
  - flooding search
  - distributed Hash Tables
Solutions to lookup problem

- **centralized servers**
  - Maintain the current location of data items in a central server
  - Search complexity $O(1)$
  - Central server becomes crucial
  - Best for simple and small applications

- **flooding search**
  - Broadcast a request for an item among the nodes
  - No additional routing informations
  - High bandwidth consumption
  - Search complexity $O(N^2)$
  - Results are not guaranteed
Solutions to lookup problem

- **distributed hash tables**
  - A global view of data distributed among many nodes.
  - Mapping nodes and data items into a common address space
  - Each DHT node manages a small number of references to other nodes
  - Queries are routed via a small number of nodes to the target node
  - Load for retrieving items should be balanced equally among all nodes
  - Robust against random failure and attacks
  - Provides a definitive answer about results
Chord Algorithm – Consistent Hashing

- supports just one operation: given a key, it maps the key onto a node.

- Consistent Hashing
  - Assign each node and key an m-bit identifier using a base hash function such as SHA-1
  - Identifiers are ordered in an identifier circle modulo $2^m$ (Chord ring)
  - Key $k$ is assigned to the first node whose identifier is equal to or follows $k$. 
Chord Algorithm – Consistent Hashing

identifier space : m=3

<table>
<thead>
<tr>
<th>node</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

identifier circle
Simple Key Lookup

- Queries are passed around the circle via successor pointers
- Requires traversing all Nodes to find the appropriate mapping

Node 0 sends a query for key 6
Chord Algorithm – Scalable Key Location

▷ Finger Table
- Each node $n$ maintains a routing table with up to $m$ entries
- The $i^{th}$ entry in the table at node $n$ contains the identifier of the first node $s$ that succeeds $n$ by at least $2^{i-1}$ on the identifier circle. ($s = \text{successor}(n+2^{i-1})$).
- $s$ is called the $i^{th}$ finger of node $n$.

▷ Definition of variables for node $n$

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

**Finger table**

$m = 3$, each node $n$ maintains at most 3 entries

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>For.</td>
<td>start</td>
</tr>
<tr>
<td>6+2^0</td>
<td>7</td>
</tr>
<tr>
<td>6+2^1</td>
<td>0</td>
</tr>
<tr>
<td>6+2^2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>For.</td>
<td>start</td>
</tr>
<tr>
<td>0+2^0</td>
<td>1</td>
</tr>
<tr>
<td>0+2^1</td>
<td>2</td>
</tr>
<tr>
<td>0+2^2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>For.</td>
<td>start</td>
</tr>
<tr>
<td>3+2^0</td>
<td>4</td>
</tr>
<tr>
<td>3+2^1</td>
<td>5</td>
</tr>
<tr>
<td>3+2^2</td>
<td>7</td>
</tr>
</tbody>
</table>
Chord Algorithm – Scalable Key Location

Pseudocode to find the successor node of an identifier

```java
// ask node n to find id’s successor
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;

// ask node n to find id’s predecessor
n.find_predecessor(id)
  n' = n;
  while (id \notin (n', n'.successor])
    n' = n'.closest_preceding_finger(id);
  return n';

// return closest_finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if (finger[i].node \in (n, id))
      return finger[i].node;
  return n;
```

Find id’s successor by finding the immediate predecessor of the id.

Walk clockwise to find the node which precedes id and whose successor succeeds id.

Start with the m\textsuperscript{th} finger of node n. See if it comes between node n and the id, if not, check the m-1\textsuperscript{th} finger until we find one which does. This is the closest node preceding id among all the fingers of n.
### Chord Algorithm – Scalable Key Location

**id=5**

**n=7**

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Int.</td>
</tr>
<tr>
<td>0</td>
<td>[0,1)</td>
</tr>
<tr>
<td>1</td>
<td>[1,3)</td>
</tr>
<tr>
<td>3</td>
<td>[3,7)</td>
</tr>
</tbody>
</table>

Successor: 0  
Predecessor: 4

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Int.</td>
</tr>
<tr>
<td>5</td>
<td>[5,6)</td>
</tr>
<tr>
<td>6</td>
<td>[6,0)</td>
</tr>
<tr>
<td>0</td>
<td>[0,4)</td>
</tr>
</tbody>
</table>

Successor: 7  
Predecessor: 3

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Int.</td>
</tr>
<tr>
<td>1</td>
<td>[1,2)</td>
</tr>
<tr>
<td>2</td>
<td>[2,4)</td>
</tr>
<tr>
<td>4</td>
<td>[4,0)</td>
</tr>
</tbody>
</table>

Successor: 3  
Predecessor: 7

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Int.</td>
</tr>
<tr>
<td>4</td>
<td>[4,5)</td>
</tr>
<tr>
<td>5</td>
<td>[5,7)</td>
</tr>
<tr>
<td>7</td>
<td>[7,3)</td>
</tr>
</tbody>
</table>

Successor: 4  
Predecessor: 0

\[ \text{successor}(5) = 7 \]

\[ O(\log N) \]
Chord Algorithm - Node joins

▷ Invariants to be preserve
  ▪ Each node’s successor is correctly maintained
  ▪ For every key k, node successor(k) is responsible for k

▷ It is desirable for the finger tables to be correct

▷ Tasks to be performed by Chord
  ▪ Initialize the predecessor and fingers of node n
  ▪ Update the fingers and predecessor of existing nodes to reflect the addition of n
  ▪ Notify the higher layer software so that it can transfer state associated with keys that node n is now responsible for
Chord Algorithm - Node joins

Initializing fingers and predecessor

```
find_successor(6);
```

finger table               keys

<table>
<thead>
<tr>
<th>start</th>
<th>Int.</th>
<th>Succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,1)</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>[1,3)</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>[3,7)</td>
<td>3</td>
</tr>
</tbody>
</table>

Successor: 0
Predecessor: 5

finger table               keys

<table>
<thead>
<tr>
<th>start</th>
<th>Int.</th>
<th>Succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1,2)</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>(2,4)</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>(4,0)</td>
<td>7</td>
</tr>
</tbody>
</table>

Successor: 3
Predecessor: 6

finger table               keys

<table>
<thead>
<tr>
<th>start</th>
<th>Int.</th>
<th>Succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>[6,7)</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>[7,1)</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>[1,5)</td>
<td>3</td>
</tr>
</tbody>
</table>

Successor: 7
Predecessor: 3

finger table               keys

<table>
<thead>
<tr>
<th>start</th>
<th>Int.</th>
<th>Succ.</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>(4,5)</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>(5,7)</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>(7,3)</td>
<td>7</td>
</tr>
</tbody>
</table>

Successor: 6
Predecessor: 0
Chord Algorithm - Node joins

Updating fingers of existing nodes

\[ P = \text{find_predecessor}(n-2^i-1) \]

\[
\begin{array}{c|c|c}
\text{start} & \text{Int.} & \text{Succ.} \\
0 & [0,1) & 0 \\
1 & [1,3) & 0 \\
3 & [3,7) & 3 \\
\end{array}
\]

Successor: 0
Predecessor: 5

\[ O(\log^2 N) \]
Chord Algorithm - Node joins

Transferring Keys

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Int.</td>
</tr>
<tr>
<td>0</td>
<td>[0,1)</td>
</tr>
<tr>
<td>1</td>
<td>[1,3)</td>
</tr>
<tr>
<td>3</td>
<td>[3,7)</td>
</tr>
</tbody>
</table>

Successor: 0
Predecessor: 5

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Int.</td>
</tr>
<tr>
<td>1</td>
<td>[1,2)</td>
</tr>
<tr>
<td>2</td>
<td>[2,4)</td>
</tr>
<tr>
<td>4</td>
<td>[4,0)</td>
</tr>
</tbody>
</table>

Successor: 3
Predecessor: 6

<table>
<thead>
<tr>
<th>finger table</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Int.</td>
</tr>
<tr>
<td>4</td>
<td>[4,5)</td>
</tr>
<tr>
<td>5</td>
<td>[5,7)</td>
</tr>
<tr>
<td>7</td>
<td>[7,3)</td>
</tr>
</tbody>
</table>

Successor: 5
Predecessor: 0
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
    if (n')
        init_finger_table(n');
        update.others();
        // move keys in (predecessor, n] from successor
    else // n is the only node in the network
        for i = 1 to m
            finger[i].node = n;
            predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
    finger[1].node = n'.find_successor(finger[1].start);
    predecessor = successor.predecessor;
    successor.predecessor = n;
    for i = 1 to m - 1
        if (finger[i + 1].start ∈ [n, finger[i].node))
            finger[i + 1].node = finger[i].node;
        else
            finger[i + 1].node =
                n'.find_successor(finger[i + 1].start);

// update all nodes whose finger
// tables should refer to n.
    n.update.others()
        for i = 1 to m
            // find last node p whose i-th finger might be n
            p = find.predecessor(n - 2^i - 1);
            p.update_finger_table(n, i);

// if s is i-th finger of n, update n's finger table with s
    n.update_finger_table(s, i)
        if (s ∈ [n, finger[i].node])
            finger[i].node = s;
            p = predecessor; // get first node preceding n
            p.update.finger_table(s, i);

Pseudocode for the node join operation
Chord Algorithm - Stabilization

- Stabilization
  - Correctness and performance
  - Keep node’s successor pointers up to date
  - Use successor pointers to verify correct finger table entries
Chord Algorithm - Stabilization

Pseudocode for stabilization

\[ n.\text{join}(n') \]
\[ \text{predecessor} = \text{nil}; \]
\[ \text{successor} = n'.\text{find-successor}(n); \]

// periodically verify n's immediate successor;
// and tell the successor about n.
\[ n.\text{stabilize}() \]
\[ x = \text{successor.predecessor}; \]
\[ \text{if } (x \in (n, \text{successor})) \]
\[ \text{successor} = x; \]
\[ \text{successor.notify}(n); \]

// \( n' \) thinks it might be our predecessor.
\[ n.\text{notify}(n') \]
\[ \text{if } (\text{predecessor is nil or } n' \in (\text{predecessor}, n)) \]
\[ \text{predecessor} = n'; \]

// periodically refresh finger table entries.
\[ n.\text{fix-fingers}() \]
\[ i = \text{random index} > 1 \text{ into finger}[]; \]
\[ \text{finger}[i].\text{node} = \text{find-successor(finger}[i].\text{start}); \]

Join does not make the rest of the network aware of n

Every node runs stabilize periodically, to verify the successor

Node n asks its successor for the successor’s predecessor x. See if x should be n’s successor instead. (happens if x recently joined the system)

Notify n’s successor of n’s exist. Successor changes its predecessor to n if it knows no closer predecessor than n.

Use successor pointers to update finger tables.
Chord Algorithm – Node Failure

- Node Failure
  - Successor-list
  - If successor fails, replaces it with the first live entry in the list
  - Later run stabilize to correct finger table and successor-list
Summary

▷ Characteristics of Chord
  ▪ Load balance
distributed hash table
  ▪ Decentralization
fully distributed
  ▪ Scalability
cost of lookup grows logarithmic
  
<table>
<thead>
<tr>
<th>Routing Hops</th>
<th>$O(\log N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival</td>
<td>$O(\log^2 N)$</td>
</tr>
<tr>
<td>Departure</td>
<td>$O(\log^2 N)$</td>
</tr>
</tbody>
</table>

  ▪ Availability
automatically adjusts internal tables
  ▪ Flexible naming
no constrains on the structure of the keys
References


▷ http://www.wikipedia.org

▷ http://www.google.com
Thank You

Questions?