Distributed Hash Tables

Chord

Smruti R. Sarangi

Department of Computer Science
Indian Institute of Technology
New Delhi, India
Outline

1. Overview

2. Design of Chord
   - Basic Structure
   - Algorithm to find the Successor
   - Node Arrival and Stabilization

3. Results
Comparison with Pastry

Chord vs Pastry

- Each node and each key’s id is hashed to a unique value.
- The process of lookup tries to find the immediate successor to a key’s id.
- The routing table at each node contains $O(\log(n))$ entries.
- Inserting and deleting nodes requires $O(\log(n)^2)$ messages.
- Sarangi View 😊: More robust than Pastry, and more elegant.
The **Globe** system assigns objects to locations, and is hierarchical. Chord is completely distributed and decentralized.

**CAN**
- Uses a d-dimensional co-ordinate space.
- Each node maintains $O(d)$ state, and the lookup cost is $O(dN^{1/d})$.
- Maintains a lesser amount of state than Chord, but has a higher lookup cost.
Features of Chord

- Automatic load balancing
- Fully distributed
- Scalable in terms of state per node, bandwidth, and lookup time.
- Always available
- Provably correct.
Outline

1. Overview

2. Design of Chord
   - Basic Structure
   - Algorithm to find the Successor
   - Node Arrival and Stabilization

3. Results
Consistent Hashing

Definition

**Consistent Hashing:** It is a hashing technique that adapts very well to resizing of the hash table. Typically \( \frac{k}{n} \) elements need to be reshuffled across buckets. \( k \) is the number of keys and \( n \) is the number of slots in a hash table.
Structure of Chord

- Each node and key is assigned a $m$ bit identifier.
- The hash for the node and key is generated by using the SHA-1 algorithm.
- The nodes are arranged in a circle (recall Pastry).
- Each key is assigned to the smallest node id that is larger than it. This node is known as the **successor**.

**Objective**

- For a given key, efficiently locate its successor.
- Efficiently manage addition and deletion of nodes.
Properties of Chord’s Hashing Algorithm

- For $n$ nodes, and $k$ keys, with high probability:
  1. Each node stores at most $(1 + \epsilon)k/n$ keys
  2. Addition and deletion of nodes leads to a reshuffling of $O(k/n)$ keys

- Previous papers prove that $\epsilon = O(\log(n))$

- There are techniques to reduce $\epsilon$ using virtual nodes.
  - Each node contains $\log(n)$ virtual nodes.
  - Not scalable (Not necessarily required)
Let $m$ be the number of bits in an id

- Node $n$ contains $m$ entries in its finger table.
  - successor $\rightarrow$ next node on the identifier circle
  - predecessor $\rightarrow$ node on the identifier circle
- The $i^{th}$ finger contains:
  - $\text{finger}[i].\text{start} = (n + 2^{i-1}) \mod 2^m, (1 \leq i \leq m)$
  - $\text{finger}[i].\text{end} = (n + 2^i - 1) \mod 2^m$
  - $\text{finger}[i].\text{node} = \text{successor}(\text{finger}[i].\text{start})$

**Basic Operation**

`findSuccessor(keyId) → nodeId`
Outline

1. Overview

2. Design of Chord
   - Basic Structure
   - Algorithm to find the Successor
   - Node Arrival and Stabilization

3. Results
Algorithms

**Algorithm 1:** findSuccessor in Chord

```plaintext
1 n.findSuccessor(id) begin
2   n' ← findPredecessor(id)
3   return n'.successor(id)
4 end
5 n.findPredecessor(id) begin
6   n' ← n
7   while id ∉ (n', n'.successor()) do
8     n' ← n'.closestPrecedingFinger(id)
9   end
10 end
```
closestPrecedingFinger(id)

1 n.closestPrecedingFinger(id) begin
2 for i ← m to 1 do
3     if finger[i].node ∈ (n, id) then
4         return finger[i].node
5     end
6 end
7 return n
8 end
Overview
Design of Chord
Results

Basic Structure
Algorithm to find the Successor
Node Arrival and Stabilization

\(O(\log(n))\) Routing Complexity

22

35

1

3

8

17 node

15

11

\(3 + 2^i\)

\(3 + 2^{i-1}\)

\(2 \cdot 2^{i-1}\)

\(< 2^{i-1}\)

predecessor

\(2^{i-1}\)
Outline

1. Overview

2. Design of Chord
   - Basic Structure
   - Algorithm to find the Successor
   - Node Arrival and Stabilization

3. Results
Each node maintains a predecessor pointer

- Initialize the predecessor and the fingers of the new node.
- Update the predecessor and fingers of other nodes
- Notify software that the node is ready
Node Arrival - II

\[ n \text{ initially contacts } n' \]

1. \( n\text{.join}(n') \) begin
2. \hspace{1em} n.initFingerTable(n')
   \hspace{1em} updateOthers()
3. end
Algorithm 2: `initFingerTable` in Chord

```plaintext
n.initFingerTable(n') begin
    finger[1].node ← n'.findSuccessor(finger[1].start)
    successor ← finger[1].node
    predecessor ← successor.predecessor
    successor.predecessor ← n
    for i ← 1 to m-1 do
        if finger[i+1].start ∈ (n, finger[i].node) then
            finger[i+1].node ← finger[i].node
        else
            finger[i+1].node ← n'.findSuccessor(finger[i+1].start)
        end
    end
end
```
```plaintext
updateOthers()

1 n.updateOthers() begin
2     for i ← 1 to m do
3         p ← findPredecessor (n - 2^{i-1})
4         p.updateFingerTable(n, i)
5     end
6 end

n.updateFingerTable(s, i) begin
7     if s ∈ (n, finger[i].node) then
8         finger[i].node ← s
9         p ← predecessor
10         p.updateFingerTable(s, i)
11     end
12 end
```
Stabilization of the Network

```plaintext
n.stabilize() begin
  x ← successor.predecessor
  if x ∈ (n, successor) then
    successor ← x
  end
  successor.notify(n)
end

n.notify(n') begin
  if (predecessor is null) OR (n' ∈ (predecessor, n)) then
    predecessor ← n'
  end
end
```
Evaluation Setup

- Network consists $10^4$ nodes
- Number of keys: $10^5$ to $10^6$
- Each experiment is repeated 20 times
- The major results are on a Chord protocol simulator
Effect of Virtual Nodes

![Bar chart showing the effect of virtual nodes on the number of keys per node.]

Source [1]
Average Path Length

![Graph showing the relationship between number of nodes and path length with 1st and 99th percentiles indicated.](source [1])
source [1]
Other DHT Systems: Tapestry

- 160 block id, Octal digits
- Routing table like pastry (digit based hypercube)
- Does not have a leaf set or neighborhood table.
Other DHT Systems: Kademlia

Kademlia

- Basis of bit-torrent
- Each node has a 128 bit id
- Each digit contains only 1 bit
- Find the closest node to a key
- Values are stored at several nodes
- Nodes can **cache** the values of popular keys.
Other DHT Systems: CAN

**CAN – Content Addressable Network**

- It uses a d-dimensional multi-torus as its overlay network.
- Node uses standard routing algorithms for tori. It uses $O(d)$ space. *(Note: This is independent of $n$)*
- Each node contains a virtual co-ordinate zone.
- Node Arrival: Split a zone
- Node Departure: Merge a zone