Distributed Hash Tables

Chord

Smruti R. Sarangi

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

Outline

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.

Comparison with other Systems

- The Globe system assigns objects to locations, and is hieararchial. 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

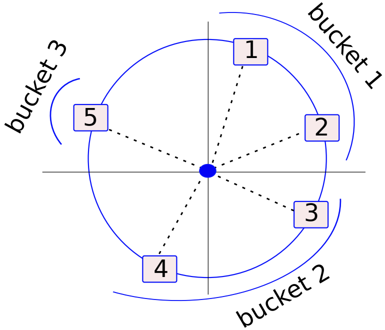
- Automatic load balancing
- Fully distributed
- Scalable in terms of state per node, bandwidth, and lookup time.
- Always available
- Provably correct.

Outline

Consistent Hashing

Definition

Consistent Hashing: It is a hashing technique that adapts very well to resizing of the hash table. Typically 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 ϵ using virtual nodes.
  - Each node contains $log(n)$ virtual nodes.
  - Not scalable (Not necessarily required)

Chord's Routing(Finger) Table

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:
  - finger[i].start = (n + 2^{i-1}) mod 2^m, $(1 \leq i \leq m)$
  - finger[i].end = (n + $2^i - 1$) mod 2^m
  - finger[i].node = successor(finger[i].start)

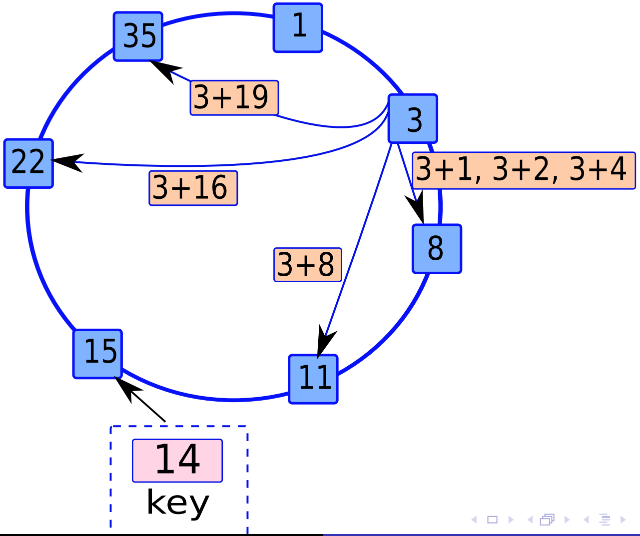
Basic Operation

findSuccessor(keyId) $\rightarrow$ nodeId

Outline

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

Finger Table- II



Algorithms

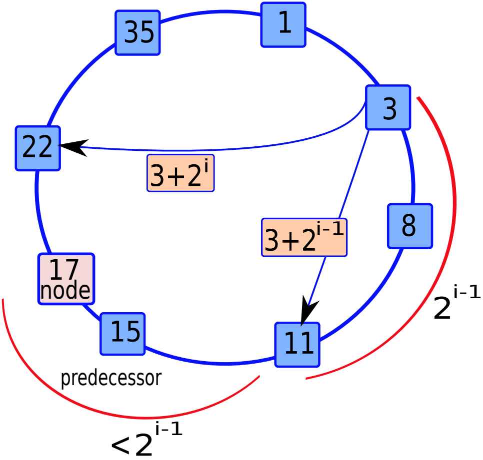
Algorithm 1: *findSuccessor* in Chord

n.findSuccessor(id) **begin**
    n' $\leftarrow$ findPredecessor(id)
    return *n'.successor(id)*
end
n.findPredecessor(id) **begin**
    $n' \leftarrow$ n
    while $id \notin (n', n'.successor())$ **do**
        n' $\leftarrow$ n'.closestPrecedingFinger(id)
    end
end

closestPrecedingFinger(id)

```
n.closestPrecedingFinger(id) begin
    for i ← m to 1 do
        if finger[i].node ∈ (n, id) then
            return finger[i].node
        end
    end
    return n
end
```

$O(log(n))$ Routing Complexity



Outline

Node Arrival

Each node maintains a precessor 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 initially contacts n'

```
n.join(n') begin
    n.initFingerTable(n')
    updateOthers()
end
```

Algorithm 2: *initFingerTable* in Chord

```
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
        end
        else
            finger[i+1].node ← n'.findSuccessor(finger[i+1].start)
        end
    end
end
```

updateOthers()

```
n.updateOthers() begin
    for i ← 1 to m do
        p ← findPredecessor (n - 2^(i−1))
        p.updateFingerTable(n, i)
    end
end
n.updateFingerTable(s, i) begin
    if s ∈ (n, finger[i].node) then
        finger[i].node ← s
        p ← predecessor
        p.updateFingerTable(s,i)
    end
end
```

Stabilization of the Netowrk

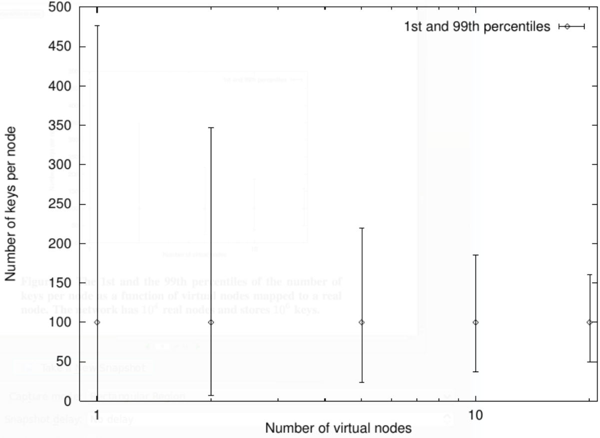
```
n.stabilize() begin
    x ← successor.predecessor
    if x ∈ (n, sucessor) then
        successor ← x
    end
    successor.notify(n)
end
n.notify(n') begin
    if (predecessor is null) OR (n' ∈ (predecessor, n)) then
        predecessor ← n'
    end
end
```

Results

Evaluation Setup

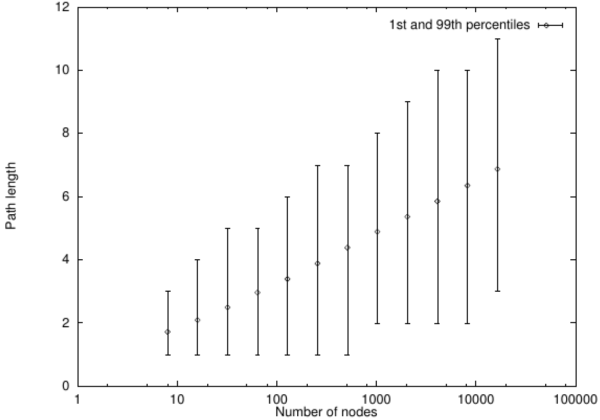
- Network consists 10^4 nodes
- Number of keys : 10^5 to 10^6
- Each experiment is repeated 20 times
- The major results are on a Chord protocol simulator

Effect of Virtual Nodes



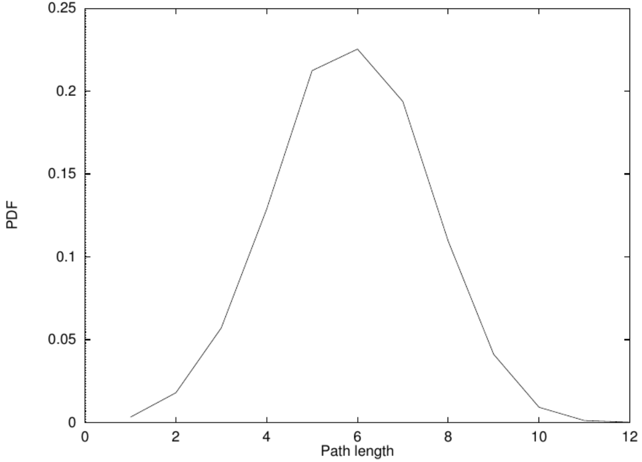
source [1]

Average Path Length



source [1]

PDF of Path Length



source [1]

Other DHT Systems: Tapestry

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

Chord: A Peer-to-Peer Lookup Service for Internet Applications, by I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, Proc. ACM SIGCOMM, San Diego, CA, September 2001.