Distributed Hash-based Lookup for Peer-to-Peer Systems

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

March 10, 2006

Agenda

- Peer-to-Peer Sytems
- Initial approaches to Peer-to-Peer Systems
- Their Limitations
- CAN - Content Addressable Network
- Applications of CAN
- Design Details
- Benefits

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Peer-to-Peer Systems

- Distributed and a decentralized architecture
- No centralized Server (unlike Client-Server Architecture)
- Any participating host can behave as the server

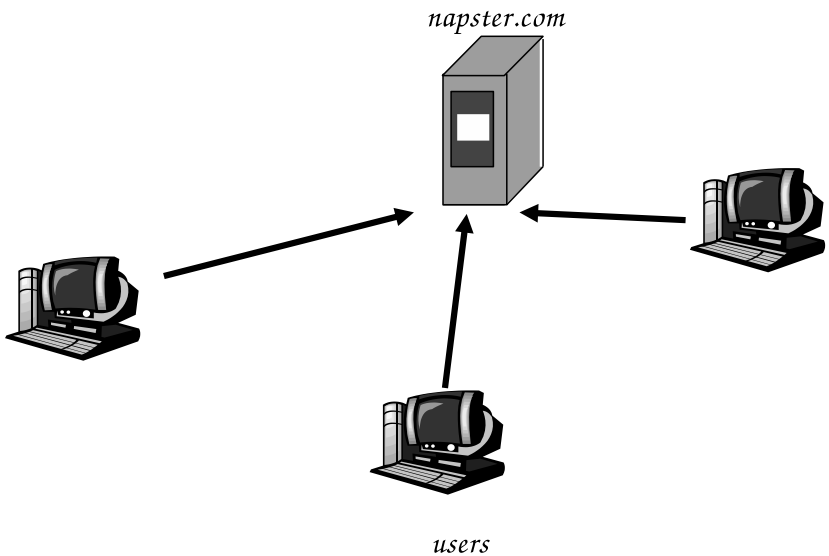
Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Napster

- P2P file sharing system
- Central Server stores the index of all the files available on the network
- To retrieve a file, central server contacted to obtain location of desired file
- Not completely decentralized system
- Central directory not scalable
- Single point of failure

Napster

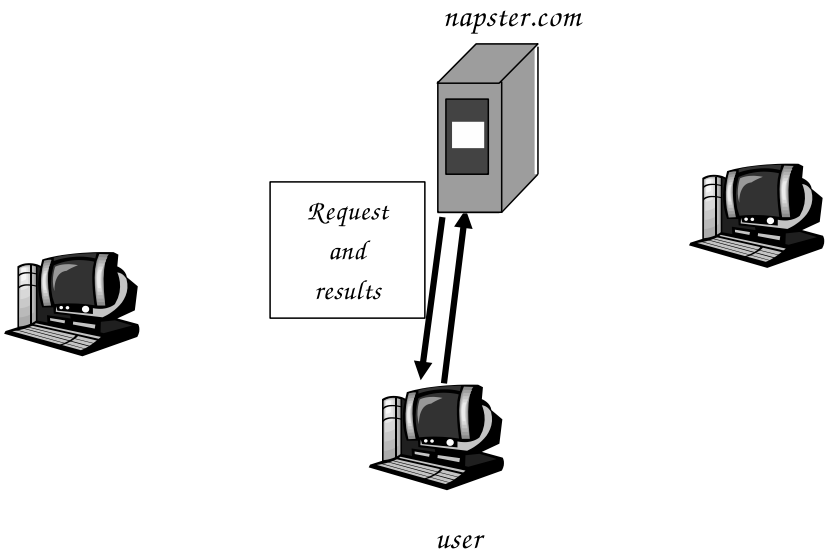
Napster



1. *File list is uploaded*

Napster

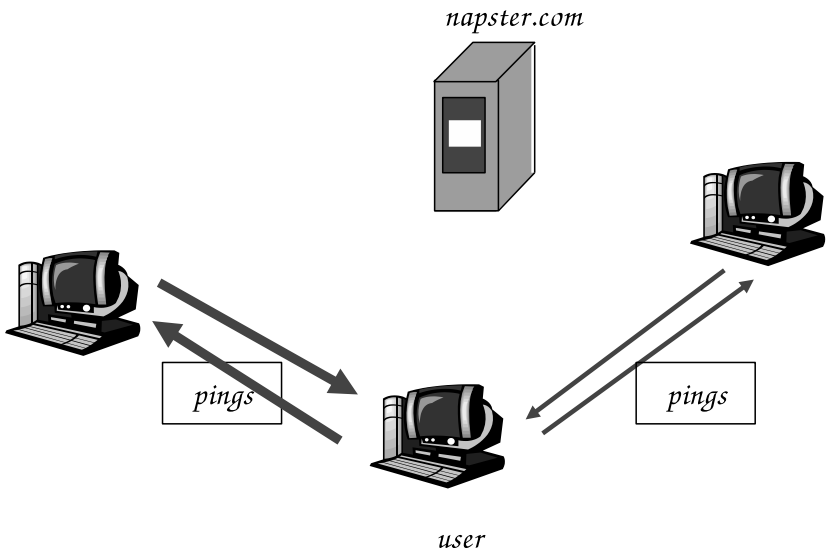
Napster



2. *User requests search at server.*

Napster

Napster

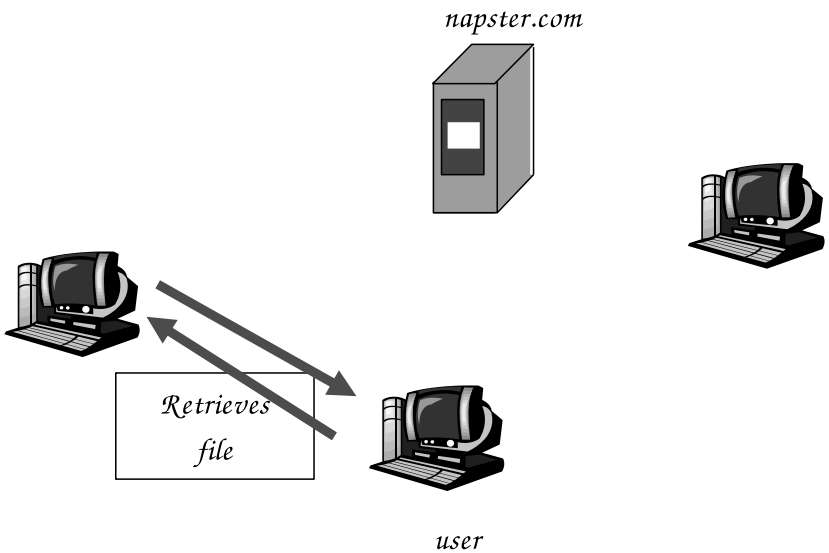


3. *User pings hosts that apparently have data.*

   Looks for best transfer rate.

Napster

Napster



4. *User retrieves file*

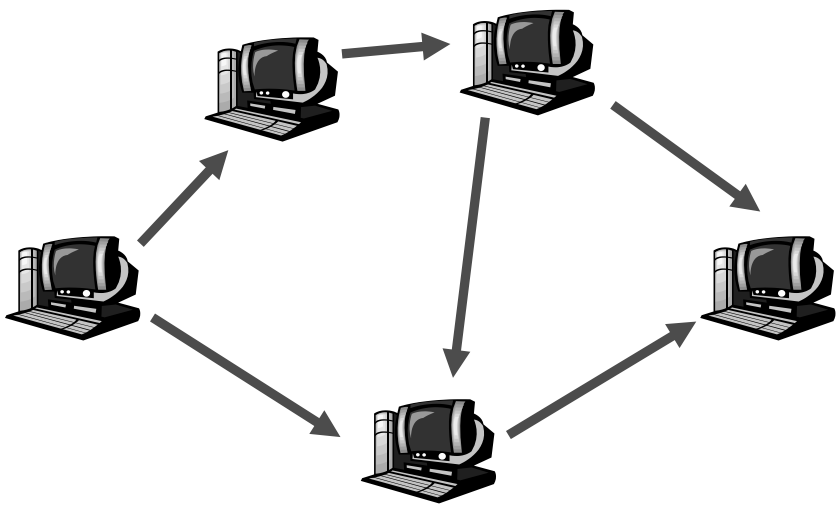
Gnutella

- P2P file sharing system
- No Central Server store to index the files available on the network
- File location process decentralized as well
- Requests for files are flooded on the network
- No Single point of failure
- Flooding on every request not scalable

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Gnutella

Flooding in Gnutella: loop prevention



Seen already list: "A"

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Content Addressable Network (CAN)

- In any P2P system, File transfer process is inherently scalable
- However, the indexing scheme which maps file names to location crucial for scalablility
- *Content Addressable Network (CAN)* provides a scalable indexing mechanism

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Applications of CAN

DNS

- Conventional DNS lookup system
  - Requires special root servers
  - Closely ties naming scheme to the manner in which a name is resolved to an IP address
- DNS lookup system using hash based lookup
  - CAN can provide DNS lookup system without any special servers
  - naming scheme independent of the name resolution process

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Applications of CAN (..continued)

File Systems for P2P systems

- The file system would store files and their meta data across nodes in the P2P network
- The nodes containing blocks of files could be located using hash based lookup
- The blocks would then be fetched from those nodes

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

What is CAN?

- CAN is a distributed infrastructure that *provides hash table like functionality*
- CAN is composed of many individual nodes
- Each CAN node stores a chunk(zone) of the entire hash table
- Request for a particular key is routed by intermediate CAN nodes whose zone contains that key
- The design can be implemented in application level *(no changes to kernel required)*

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Design of CAN

- Involves a *virtual d-dimensional Cartesian Co-ordinate space*
- The co-ordinate space is completely *logical*
- The co-ordinate space is partitioned into zones among all nodes in the system
- Every node in the system owns a distinct zone
- The distribution of zones into nodes forms an *overlay network*

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Co-ordinate space in CAN

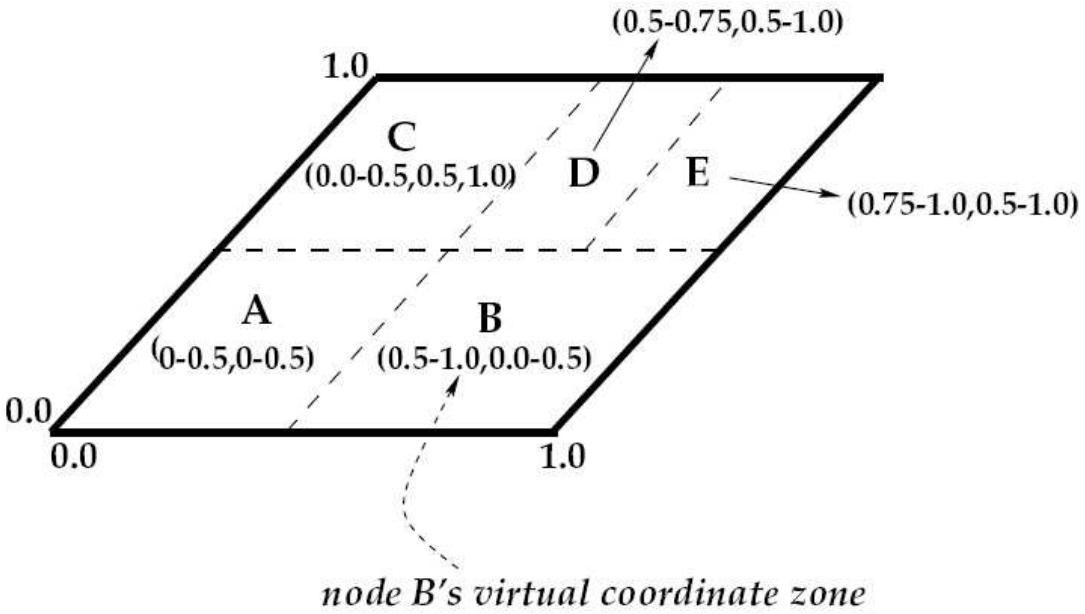


Figure 1: ***Example 2-d coordinate overlay with 5 nodes***

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Design of CAN (..continued)

- To store *(Key,value)* pairs, keys are mapped deterministically onto a point P in co-ordinate space using a hash function
- The *(Key,value)* pair is then stored at the node which owns the zone containing P
- To retrieve an entry corresponding to Key K, the same hash function is applied to map K to the point P
- The retrieval request is routed to node owning zone containing P

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Routing in CAN

- Every CAN node holds IP address and virtual co-ordinates of each of it's neighbours
- Every message to be routed holds the destination co-ordinates
- Using it's neighbour's co-ordinate set, a node routes a message towards the neighbour with co-ordinates closest to the destination co-ordinates
- Even if one of the neighbours fails, messages can be routed through other neighbours in that direction
- How close the message gets closer to the destination after being routed to one of the neighbours is called *Progress*

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Routing in CAN (..continued)

- For a d-dimensional space partitioned into n equal zones, routing path length $= O(d.n^{1/d})$ *hops*
- Every node has *2d neighbours*
- With increase in no. of nodes, *per node state does not change*
- With increase in no. of nodes, routing path length grows as $O(n^{1/d})$

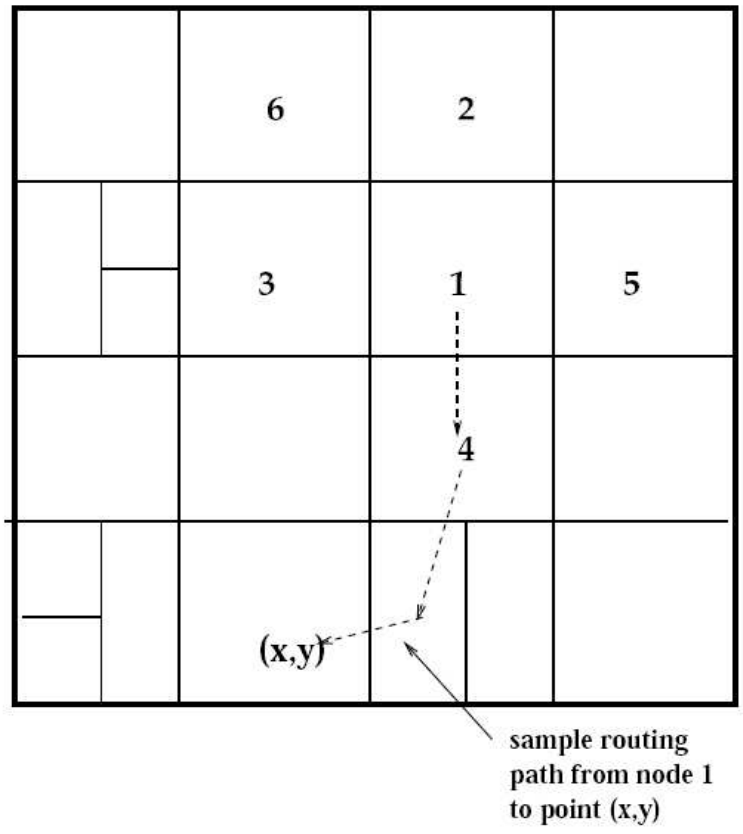
Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Allocation of a new node to a zone

- The new node randomly choses a point P in the co-ordinate space
- It sends a JOIN request to point P via any existing CAN node
- The request is forwarded using CAN routing mechanism to the node D owning the zone containing P
- D then splits it's node into half and assigns one half to new node
- The new neighbour information is determined for both the nodes

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

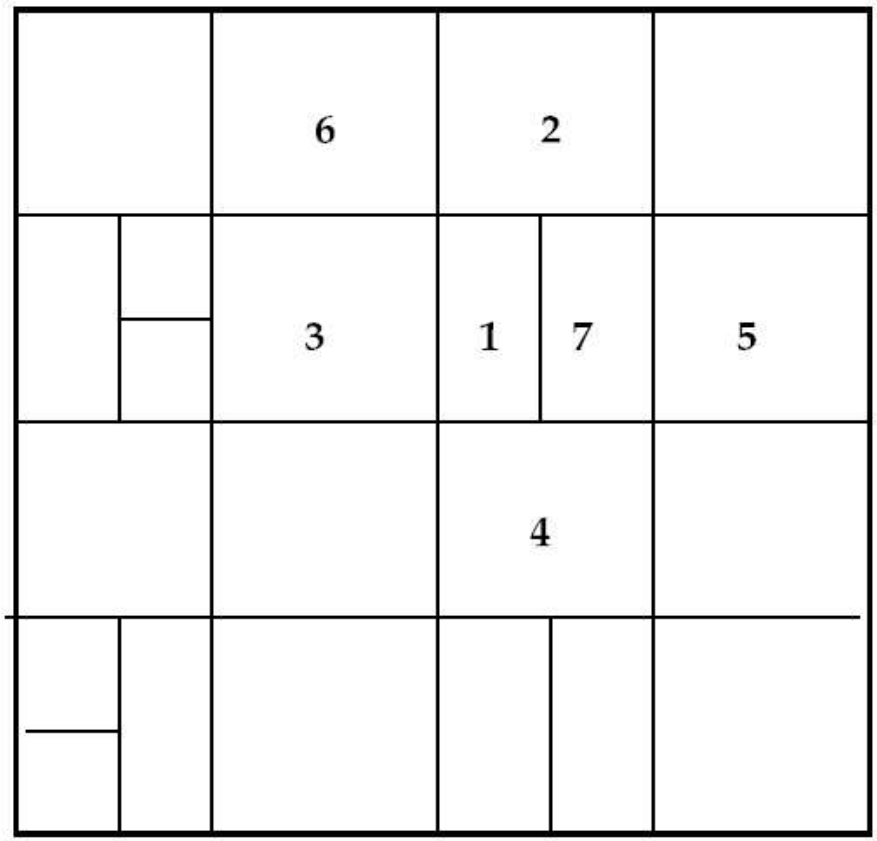
Before a node joins CAN



1's coordinate neighbor set = {2,3,4,5}
7's coordinate neighbor set = { }

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

After a node joins CAN



1's coordinate neighbor set = {2,3,4,7}
7's coordinate neighbor set = {1,2,4,5}

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Failure of a node

- If a node leaves CAN, the zone it occupies is taken over by the remaining nodes
- If a node leaves voluntarily, it can handover it's database to some other node
- When a node simply becomes unreachable, the database of the failed node is lost

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Design Improvements

- Multidimensional co-ordinate spaces
- Multiple co-ordinate spaces
- Better routing metrics
- Overloading co-ordinate zones
- Multiple hash functions
- Topologically sensitive construction of overlay network

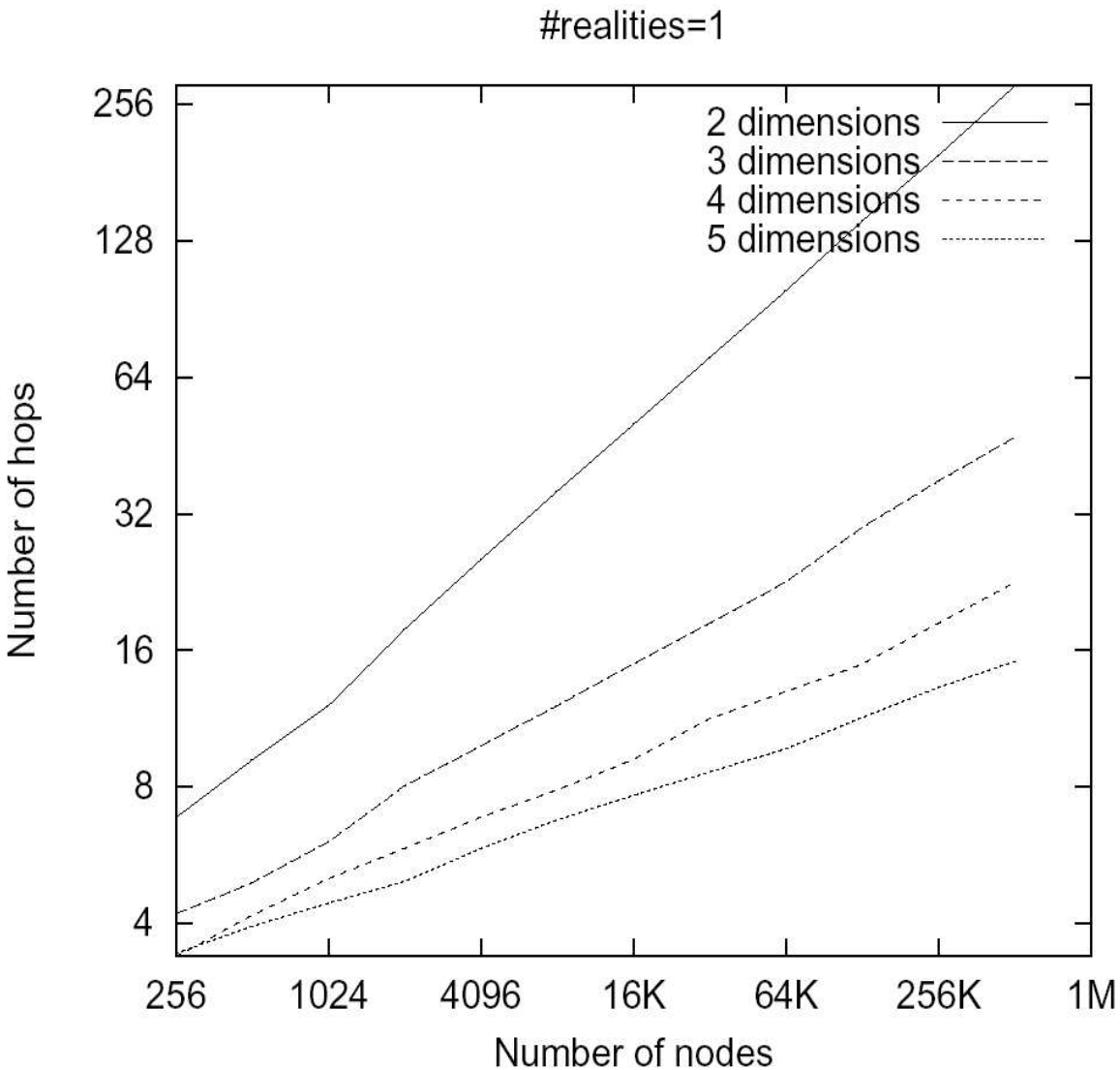
Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Multidimensional co-ordinate spaces

- Increase the dimensions of the CAN co-ordinate space
- This reduces routing path length
- As a result, path latency reduces
- Fault tolerance increases

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Multidimensional co-ordinate spaces



Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Multiple co-ordinate spaces

- Multiple, independent co-ordinate spaces (Reality) can be maintained
- Each node assigned a different zone in each co-ordinate space
- Contents of hash table replicated on each reality
- With r realities, data is unavailable only when the node corresponding to it in each of the r realities is unavailable
- Thus, improved fault tolerance
- During routing, a node forwards the message to the neighbour closest to the destination (on either reality)
- Thus, reduced path length

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Multiple co-ordinate spaces

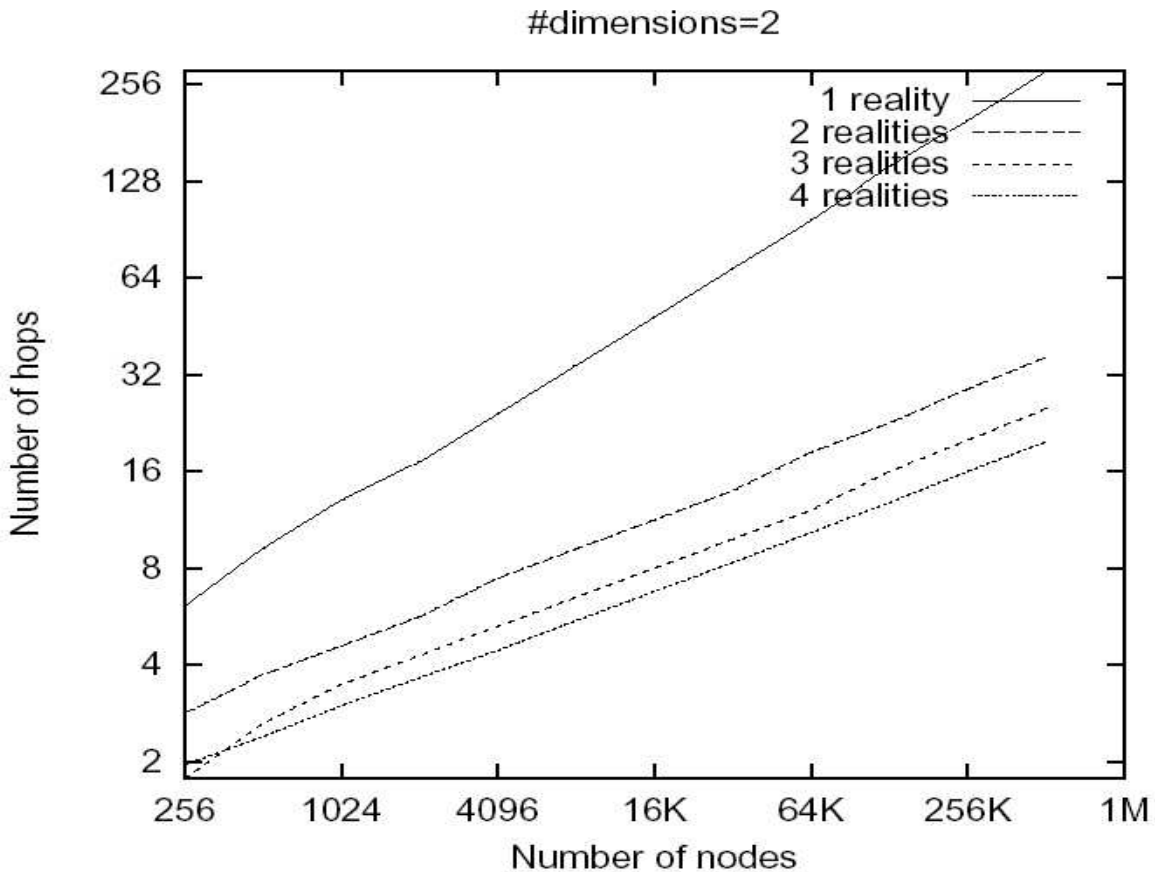


Figure 5: ***Effect of multiple realities on path length***

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Better routing metrics

- Instead of using only progress made towards the destination, use ratio of progress to RTT as a metric
- Forward to neighbour with maximum ratio of progress to RTT
- This aims at reducing the latency of individual hops

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Overloading co-ordinate zones

- Allow more than 1 node to share a zone
- The contents of the hash table can be replicated among the nodes in the same zone to increase availability
- Overloading Advantages
  - Reduced number of hops
  - Reduces per hop latency, because node can select closest node from among those in the neighbouring zones
  - Improved fault tolerance - Zone vacant only when all nodes in it fail

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Multiple hash functions

- Map each key onto nodes using *k* different hash functions
- Replicate *(Key,Value)* pair at *k* different nodes
- Queries for a particular key can then be sent to all k nodes parallelly
- However, size of (key,value) database increases
- Also, Query traffic increases by a factor of k (Improvement possible)

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Topologically sensitive construction of overlay network

- Earlier design approaches try to reduce path latency for a given overlay network
- However, they do not improve the overlay network itself
- In an overlay network, neighbouring nodes can be actually far enough
- Also, topologically close nodes can be far enough in the co-ordinate space
- Hence, need to locate topologically close nodes in nearby zones of the co-ordinate space

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Topologically sensitive construction of overlay network ..continued

- Locate m well known machines on the network
- Measure RTT of every CAN node from these m machines
- Group nodes into zones based on their relative ordering from the m machines

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

Other Design Improvements

- Uniform Partitioning
- Caching
- Replication

Benefits of CAN

- No centralized control/coordination
- Control information/state stored at nodes independent of number of nodes in the system
- Improved fault tolerance
- Scalable

Sandeep Shelke 05305402
Shrirang Shirodkar 05305007
MTech I CSE

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

March 10, 2006

Features

- Addresses a fundamental problem in Peer-to-Peer
  - Efficient location of the node that stores desired data item
  - One operation: Given a key, maps it onto a node
  - Data location by associating a key with each data item
- Adapts Efficiently
  - Dynamic with frequent node arrivals and departures
  - Automatically adjusts internal tables to ensure availability
- Uses Consistent Hashing
  - Load balancing in assigning keys to nodes
  - Little movement of keys when nodes join and leave
- Efficient Routing
  - Distributed routing table
  - Maintains information about only $O(logN)$ nodes
  - Resolves lookups via $O(logN)$ messages

Features (continued)

- Scalable
  - Communication cost and state maintained at each node scales logarithmically with number of nodes
- Flexible Naming
  - Flat key-space gives applications flexibility to map their own names to Chord keys
- Decentralized
  - Fully distributed
  - No “super” nodes

Example Applications

- Cooperative Mirroring
  - Multiple content providers spreading the load evenly
  - Low cost since each participant provides for average load
- Time-shared Storage
  - Nodes with intermittent connectivity
  - Data stored elsewhere when node is disconnected
  - Data's name serves as key to identify node
- Distributed Indexes
  - Keyword search like Napster
- Large-scale Combinatorial Search
  - Code breaking
  - Keys are candidate solutions such as cryptographic keys
  - Keys mapped to testing machines
- Chord File System (CFS)
  - Stores files and meta-data in a peer-to-peer system
  - Good lookup performance despite dynamic system

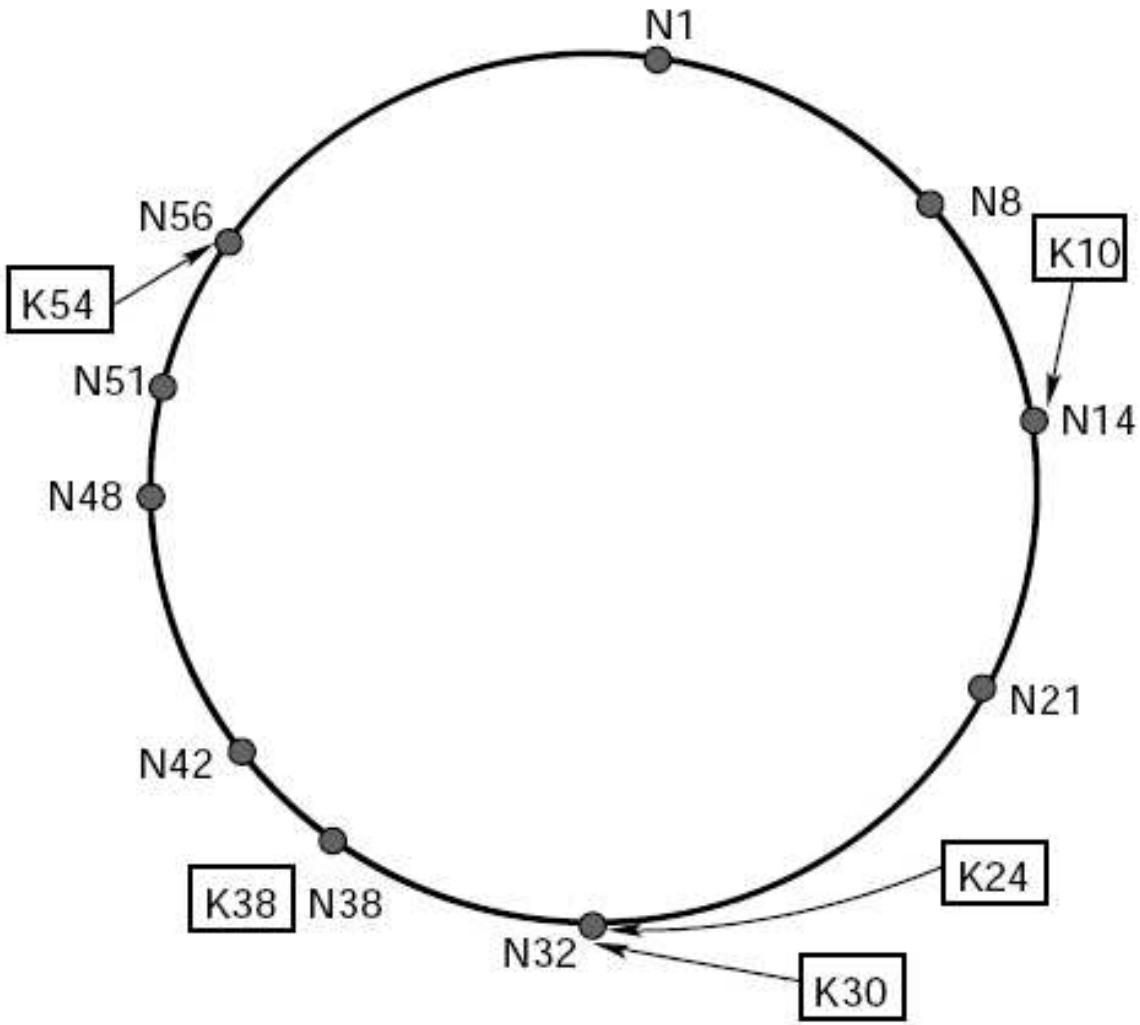
The Chord Protocol

- Consistent Hashing
- Simple Key Location
- Scalable Key Location
- Node Joins and Stabilization
- Failure and Replication

Some Terminology

- Key
  - Hash key or its image under hash function, as per context
  - m-bit identifier, using SHA-1 as a base hash function
- Node
  - Actual node or its identifier under the hash function
  - Length m such that low probability of a hash conflict
- Chord Ring
  - The identifier circle for ordering of 2^m node identifiers
- Successor Node
  - First node whose identifier is equal to or follows key k in the identifier space
- Virtual Node
  - Introduced to limit the bound on keys per node to K/N
  - Each real node runs $\Omega(logN)$ virtual nodes with its own identifier

Example: Chord Ring



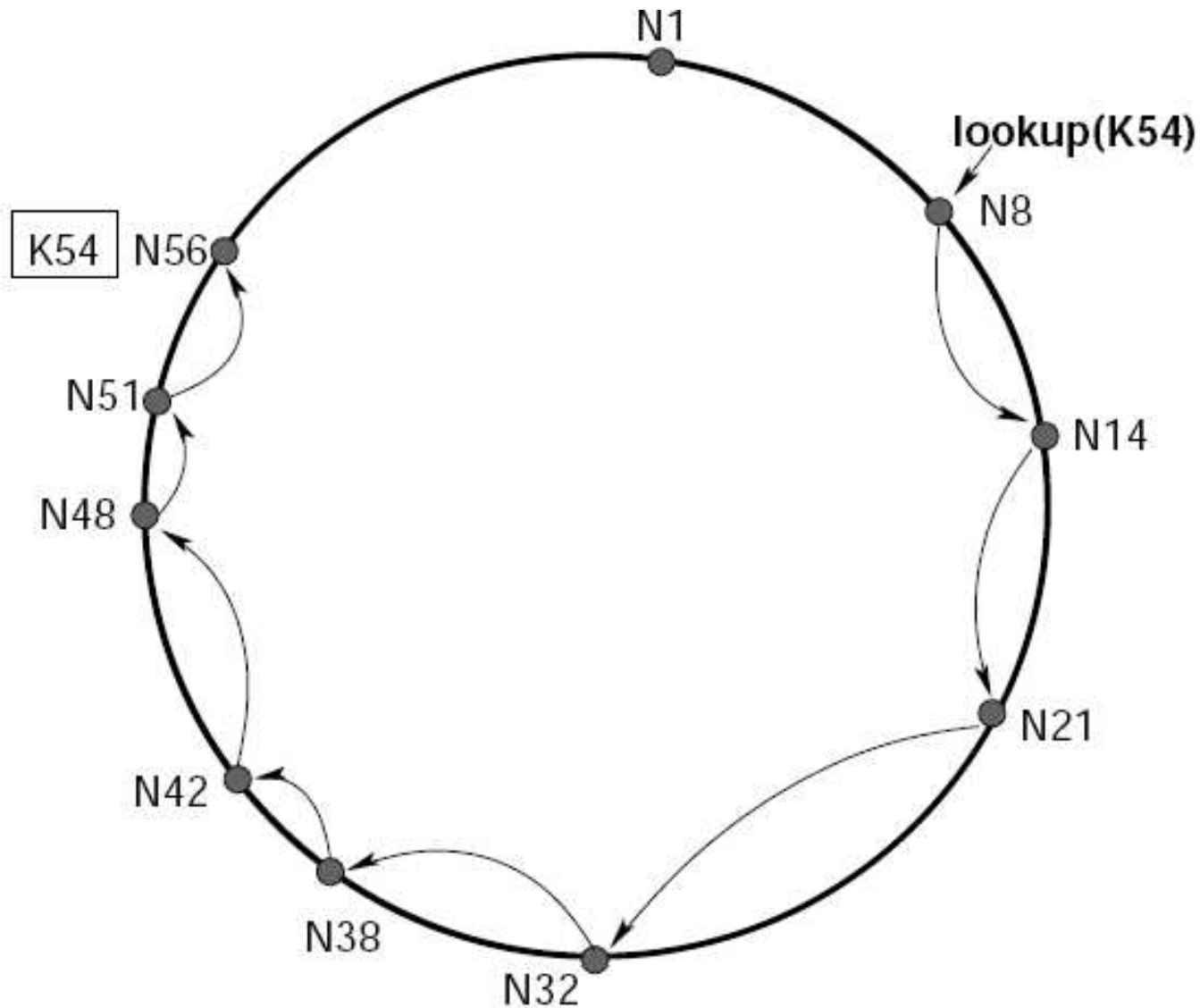
Consistent Hashing

- A consistent hash function is one which changes minimally with changes in the range of keys and a total remapping is not required
- Desirable properties
  - High probability that the hash function balances load
  - Minimum disruption, only $O(1/N)$ of the keys moved when a nodes joins or leaves
  - Every node need not know about every other node, but a small amount of "routing" information
- m-bit identifier for each node and key
- Key k assigned to Successor Node

Simple Key Location

```
// 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);
```

Example



Scalable Key Location

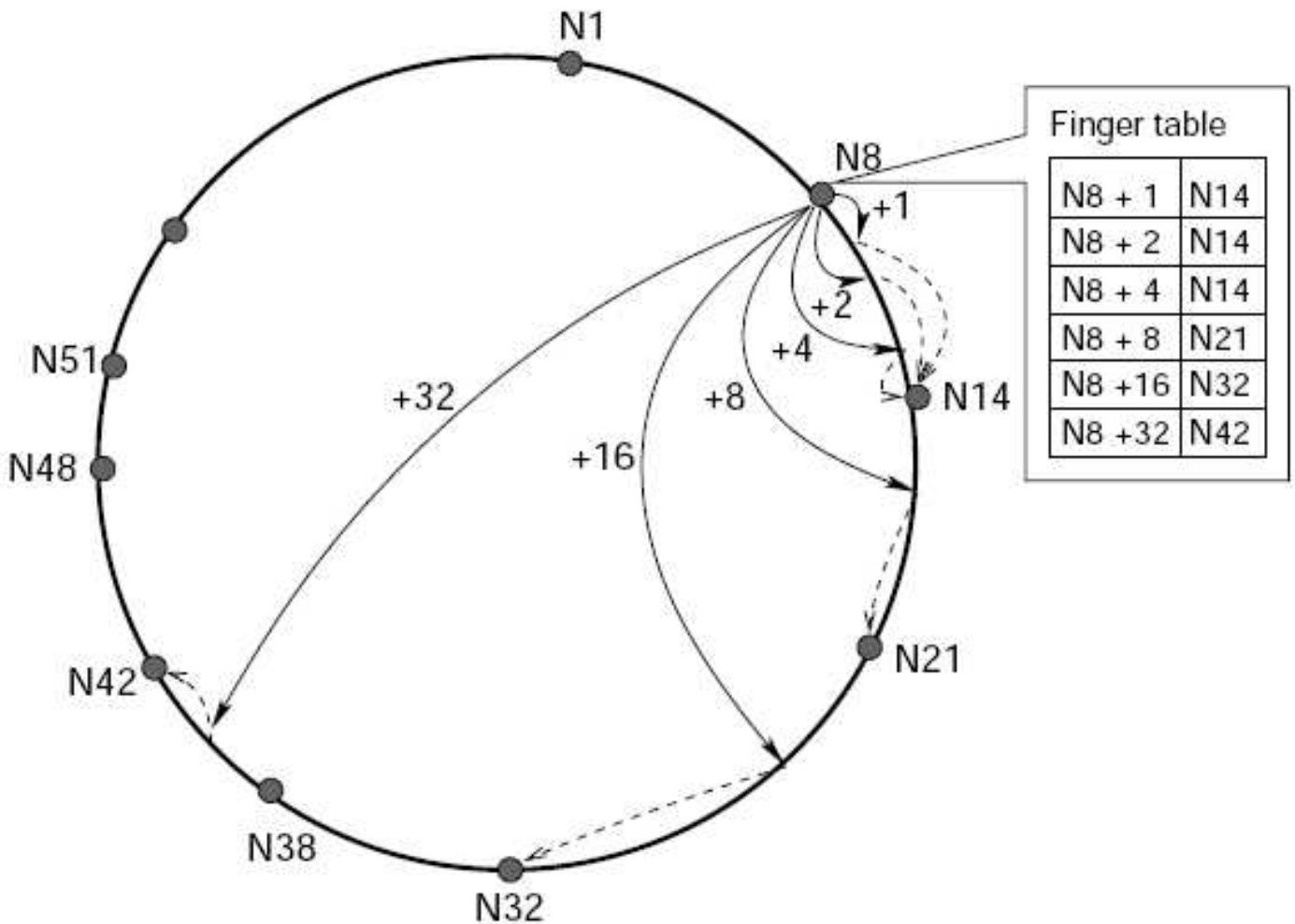
- Accelerates loopups
  - Maintains additional routing information
  - Not essential for correctness
- Finger table
  - Routing table containing upto m entries, where m is number of bits in the key/node identifier
  - i^{th} entry at node n contains identity of the first node s that succeeds n by at least 2^{i-1} on the identifier circle
  - First finger referred to as *successor*
- Each node stores information only about small number of other nodes and more about nodes closely following
- The number of nodes that must be contacted to find successor in an N-node network is $O(logN)$

Pseudocode

```
// 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;
```

Example



Stabilization

- Ensures that lookups execute correctly, as the set of participating nodes changes
- Runs periodically in the background and updates Chord's finger tables and successor pointers
- If any sequence of join operations is executed interleaved with stabilizations, then at some time after the last join the successor pointers will from a cycle on all the nodes
- In practice
  - Chord ring will never be in a stable state
  - No time to stabilize
  - "almost stable"

Pseudocode

```
// 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);
```

Pseudocode (continued)

```
// 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;
```

Impact of Node Joins on Lookups: Correctness

- For a lookup before stabilization has finished,
- Case 1: All finger table entries inolved in the lookup are reasonably current, and lookup finds correct successor in $O(logN)$ steps
- Case 2: Successor pointers are correct, but finger pointers are inaccurate. Yields correct lookups but may be slower
- Case 3: Incorrect successor pointers or keys not migrated yet to newly joined nodes. Lookup may fail
- Option of retrying after a quick pause, during which stabilization fixes successor pointers

Impact of Node Joins on Lookups: Performance

- After stabilization, no effect other than increasing the value of N in $O(logN)$
- Before stabilization is complete
- Possibly incorrect finger table entries
- Does not significantly affect lookup speed, since distance halving property depends only on ID-space distance
- If new nodes' IDs are between the target predecessor and the target, then lookup speed is influenced
- Still takes $O(logN)$ time for N new nodes

Handling Failures

- Maintain successor list of size r, containing the node's first r successors
- If immediate successor does not respond, substitute the next entry in the successor list
- Modified version of *stabilize* protocol to maintain the successor list
- Modified *closest_preceding_node* to search not only finger table but also successor list for most immediate predecessor
- Voluntary Node Departures
  - Transfer keys to successor before departure
  - Notify predecessor p and successor s before leaving

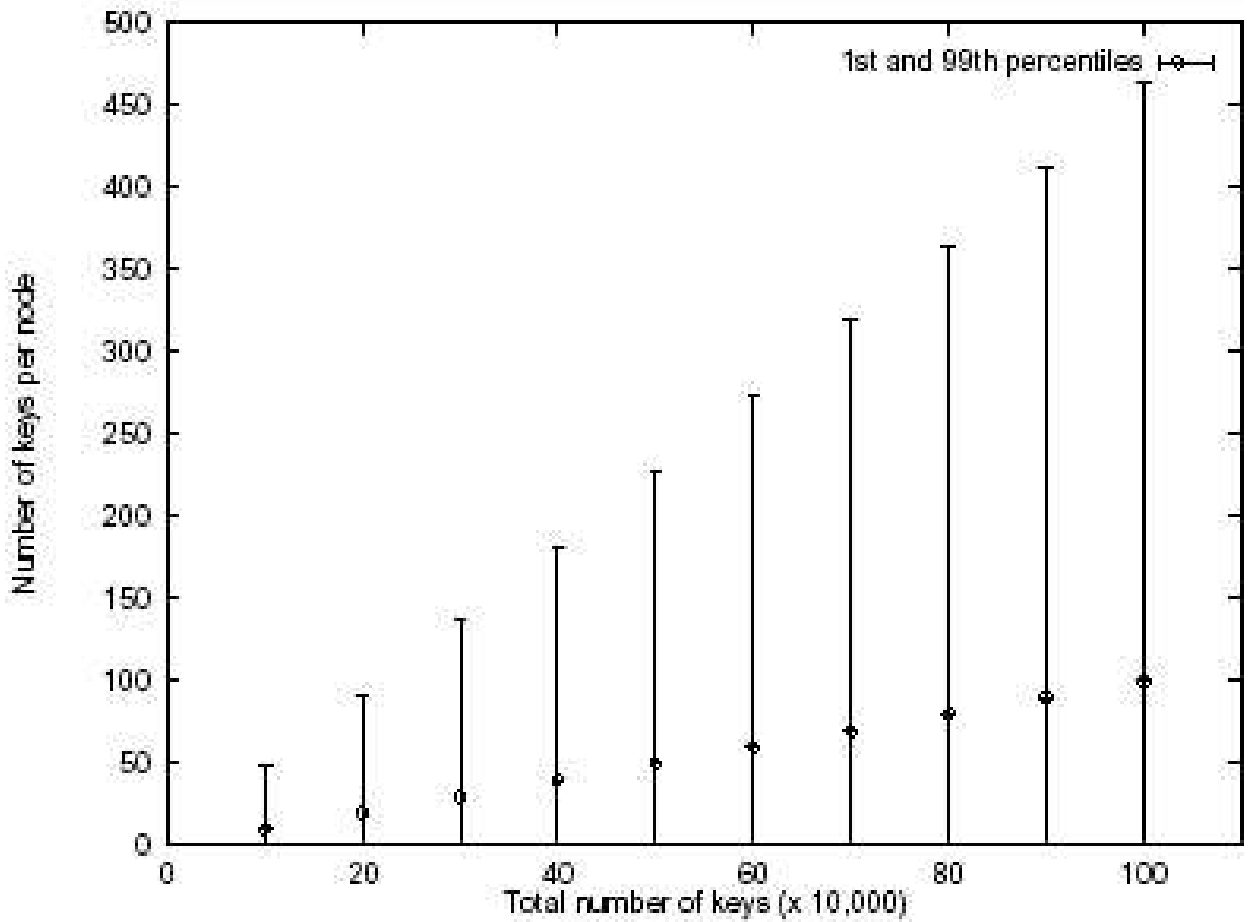
Simulation

- Implements Iterative Style
  - Node resolving a lookup initiates all communication unlike Recursive Style, where intermediate nodes forward request
- Optimizations
  - During stabilization, a node updates its immediate successor and 1 other entry in successor list or finger table
  - Each entry out of k unique entries gets refreshed once in 'k' stabilization rounds
  - Size of successor list is 1
  - Immediate notification of predecessor change to old predecessor, without waiting for next stabilization round

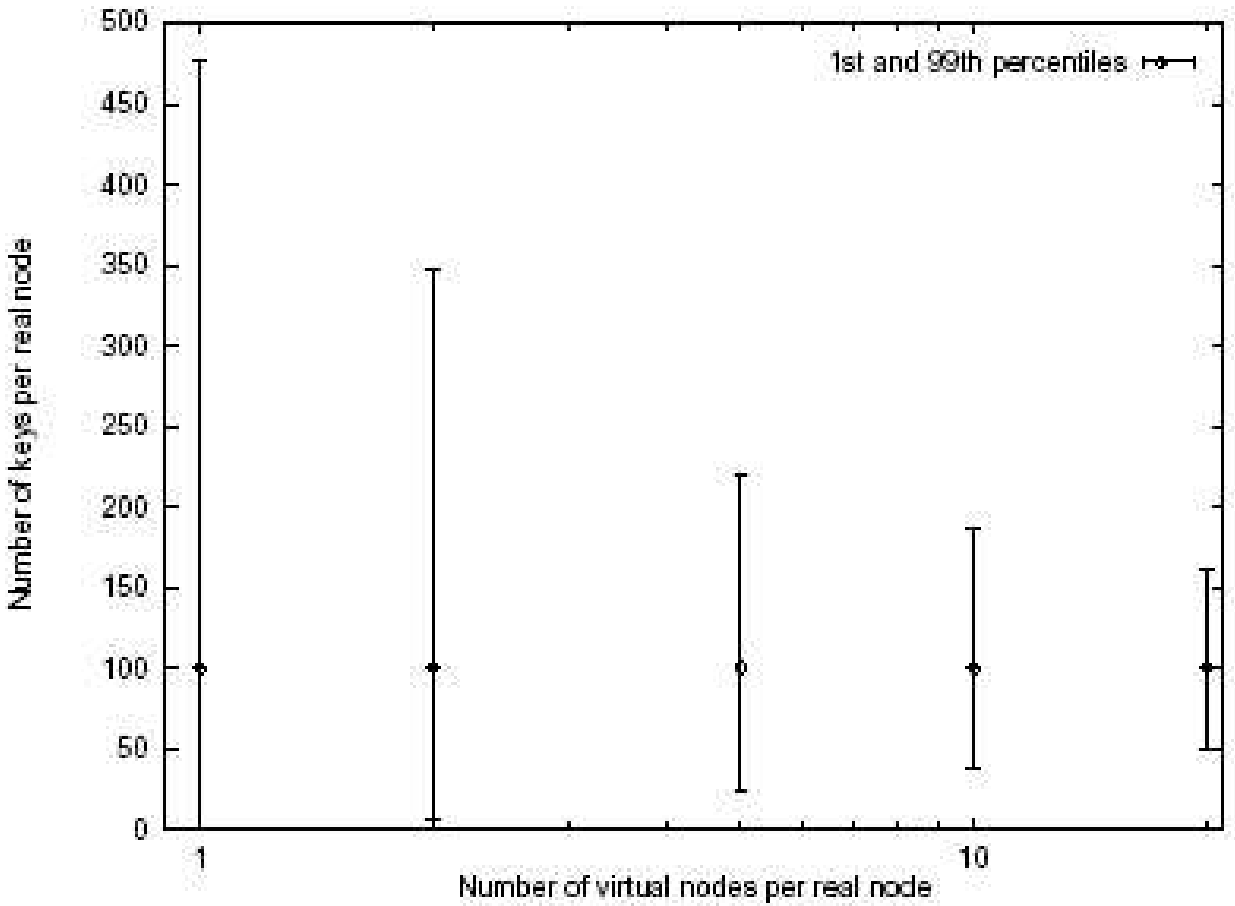
Load Balance

- Test ability of consistent hashing, to allocate keys to nodes evenly
- Number of keys per node exhibits large variations, that increase linearly with the number of keys
- Association of keys with Virtual Nodes
  - Makes the number of keys per node more uniform
  - Significantly improves load balance
  - Asymptotic value of query path length not affected much
  - Total identifier space covered remains same on average
  - Worst-case number of queries does not change
  - Not much increase in routing state maintained
  - Asymptotic number of control messages not affected

In Absence of Virtual Nodes



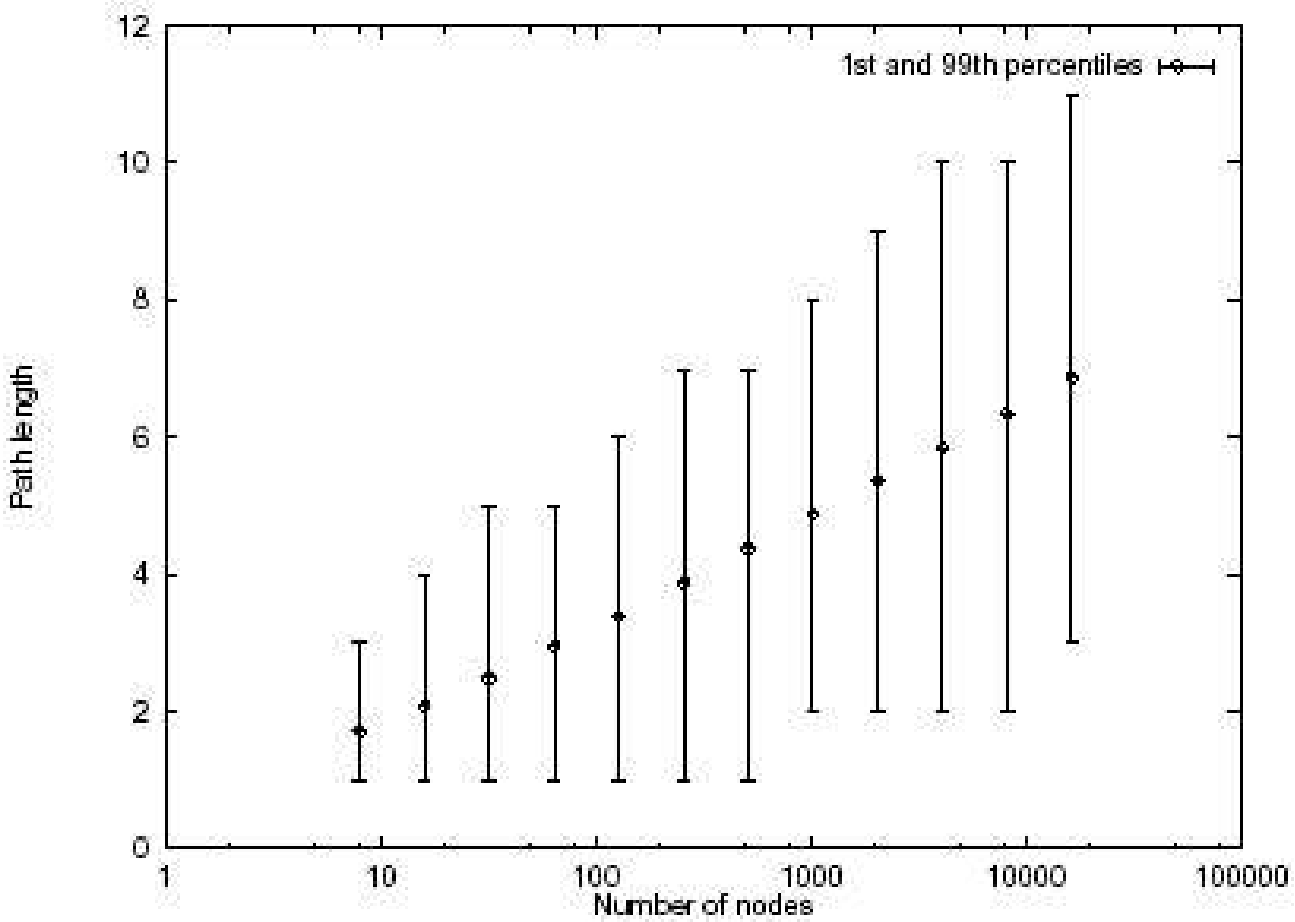
In Presence of Virtual Nodes



Path Length

- Number of nodes that must be visited to resolve a query, measured as the query path length
- As per theorem, this number is $O(logN)$, where N is the number of nodes
- Let $N = 2^k$ nodes, vary k from 3 to 14
- Each node picks random set of keys to query
- Observed Results
  - Mean query path length increases logarithmically with number of nodes
  - Same as expected average query path length

Graph



Future Work

- Resilience against Network Partitions
  - Detect and heal partitions
  - For every node have a set of initial nodes
  - Maitain a long term memory of a random set of nodes
  - Likely to include nodes from other partition
- Handle Threats to Availability of data
  - Malicious participants could present incorrect view of data
  - Periodical Global Consistency Checks for each node
- Better Efficiency
  - $O(logN)$ messages per lookup too many for some apps
  - Increase the number of fingers

Conclusion

- Decentralized Solution
- Efficient Lookup Mechanism
- Simplicity
- Provable Performance
- Provable Correctness
- A Powerful Primitive
- Applications

Thank You !