Distributed Data Structure

Zhihong Chong

http://cse.seu.edu.cn/people/zhchong/

Content

- Why Advanced Data Structures?
  - Week1: Data Structure and Computer Science, Algorithm, Database, WWW…
- Principles of Data Structures—How?
  - Week2: Why and How O(log n) access time?
  - Week3: Dynamic data structures and Analysis
  - Week4: Randomized Data Structure
  - Week5: Augmented Data Structure
  - Week6: Data Structures in Distributed Environments
- New Problems—Applications
  - Week7: Data Structures in Frontiers of Research
  - Week8: Exam

Outline

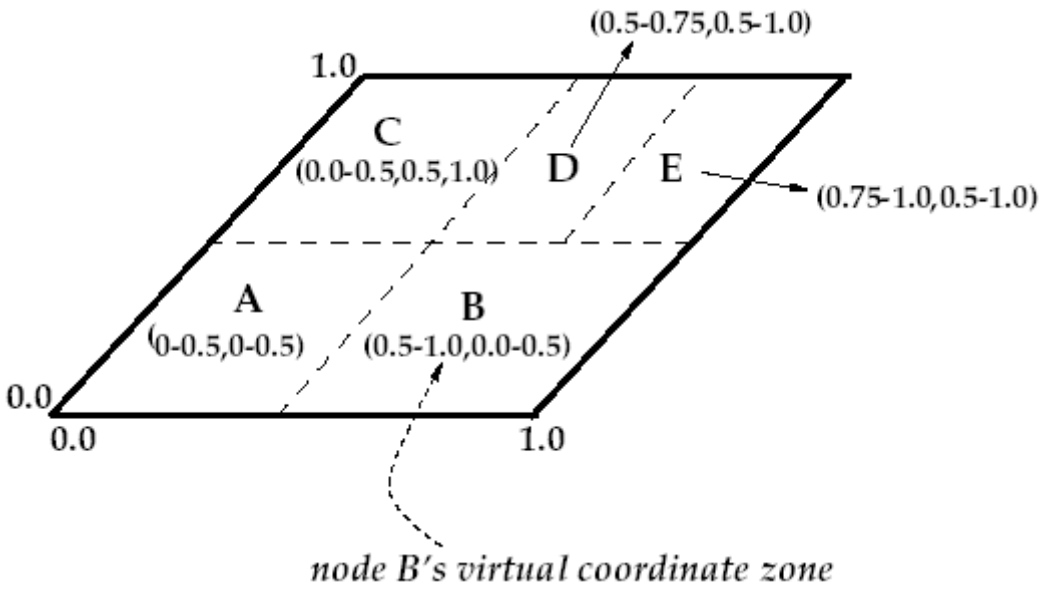
- Can
- Chord
- Baton

CAN: Overview

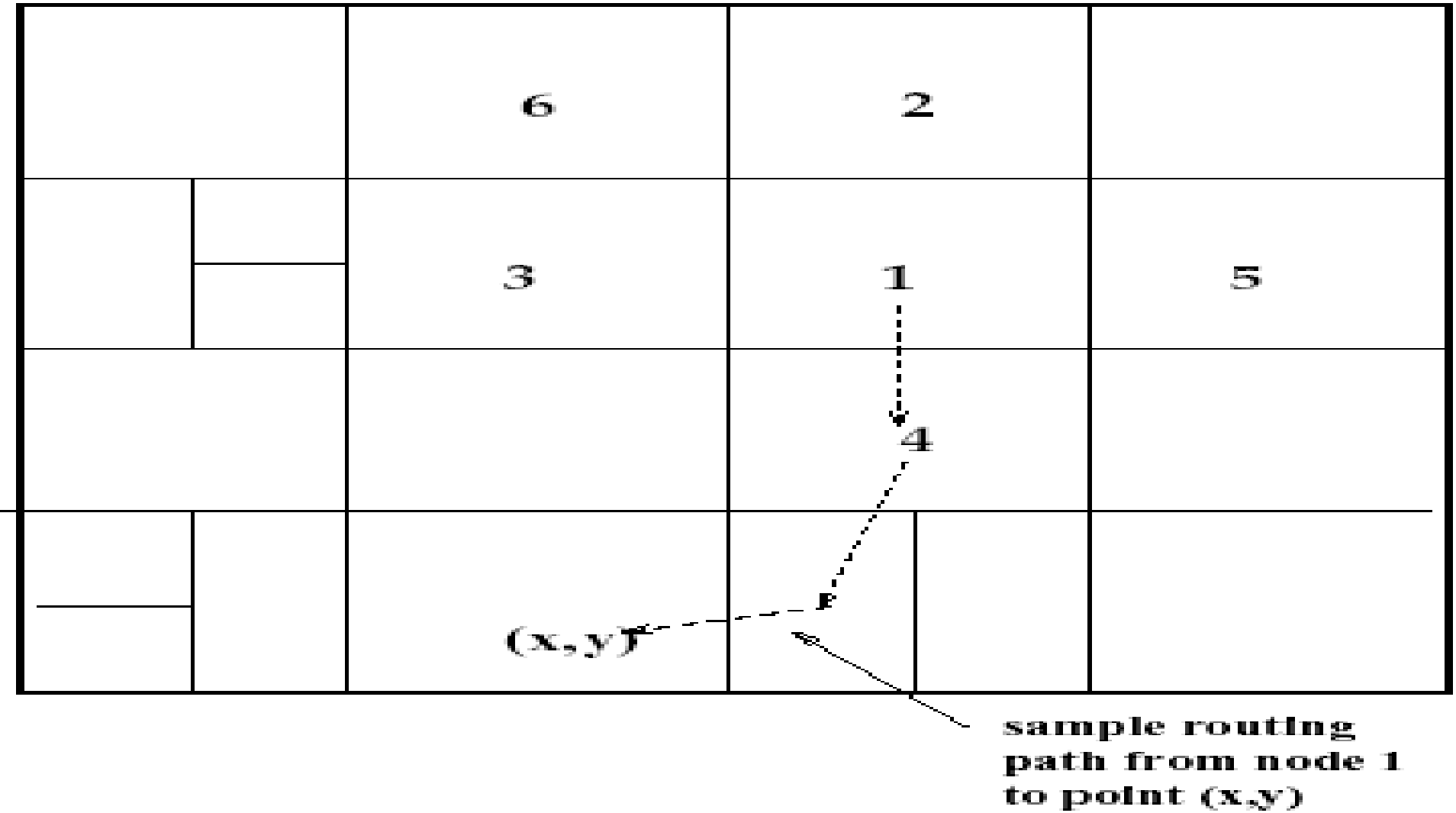
- Support basic hash table operations on key-value pairs (*K*,*V*): insert, search, delete
- CAN is composed of individual nodes
- Each node stores a chunk (zone) of the hash table
  - A subset of the (*K*,*V*) pairs in the table
- Each node stores state information about neighbor zones
- Requests (insert, lookup, or delete) for a key are routed by intermediate nodes using a greedy routing algorithm
- Requires no centralized control (completely distributed)
- Small per-node state is independent of the number of nodes in the system (scalable)
- Nodes can route around failures (fault-tolerant)

CAN Zones

- Virtual *d*-dimensional Cartesian coordinate system
  - Example: 2-*d* [0,1]x[1,0]
- Dynamically partitioned among all nodes



- Pair (*K*,*V*) is stored by mapping key *K* to a point *P* in the space using a uniform hash function and storing (*K*,*V*) at the node in the zone containing *P*
- Retrieve entry (*K*,*V*) by applying the same hash function to map *K* to *P* and retrieve entry from node in zone containing *P*
  - If P is not contained in the zone of the requesting node or its neighboring zones, route request to neighbor node in zone nearest *P*



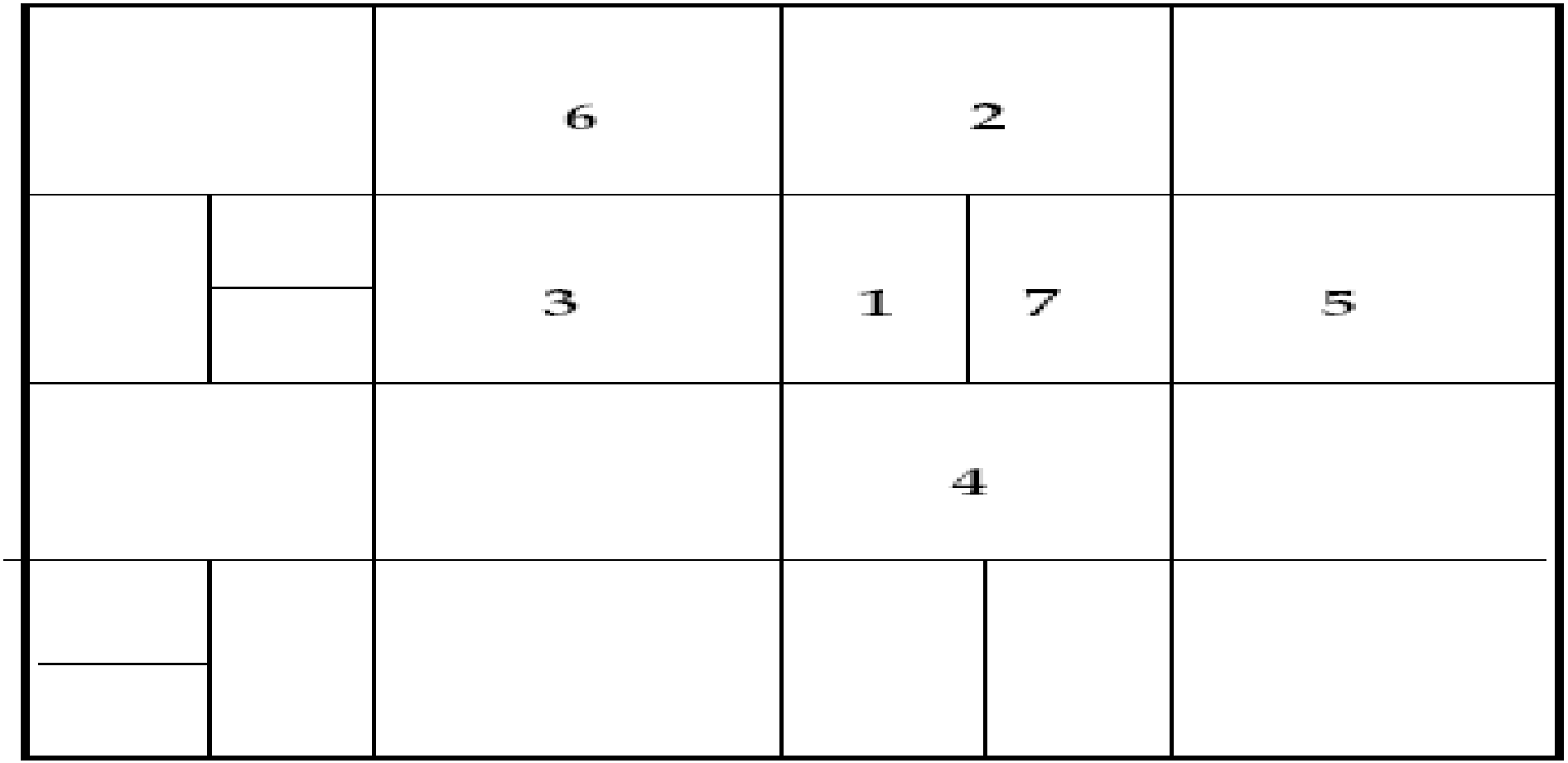
$1's\ coordinate\ neighbor\ set = \{2,3,4,5\}$
$7's\ coordinate\ neighbor\ set = \{\ \}$

Routing:

- Follow straight line path through the Cartesian space from source to destination coordinates
- Each node maintains a table of the IP address and virtual coordinate zone of each local neighbor
- Use greedy routing to neighbor closest to destination
- For *d*-dimensional space partitioned into *n* equal zones, nodes maintain 2*d* neighbors $\frac{d}{4}\left(n^{\frac{1}{d}}\right)$
  - Average routing path length:

Join CAN

- Joining node locates a bootstrap node using the CAN DNS entry
  - Bootstrap node provides IP addresses of random member nodes
- Joining node sends JOIN request to random point *P* in the Cartesian space
- Node in zone containing *P* splits the zone and allocates "half" to joining node
- (*K*,*V*) pairs in the allocated "half" are transferred to the joining node
- Joining node learns its neighbor set from previous zone occupant
  - Previous zone occupant updates its neighbor set



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

Departure, Recovery and Maintenance

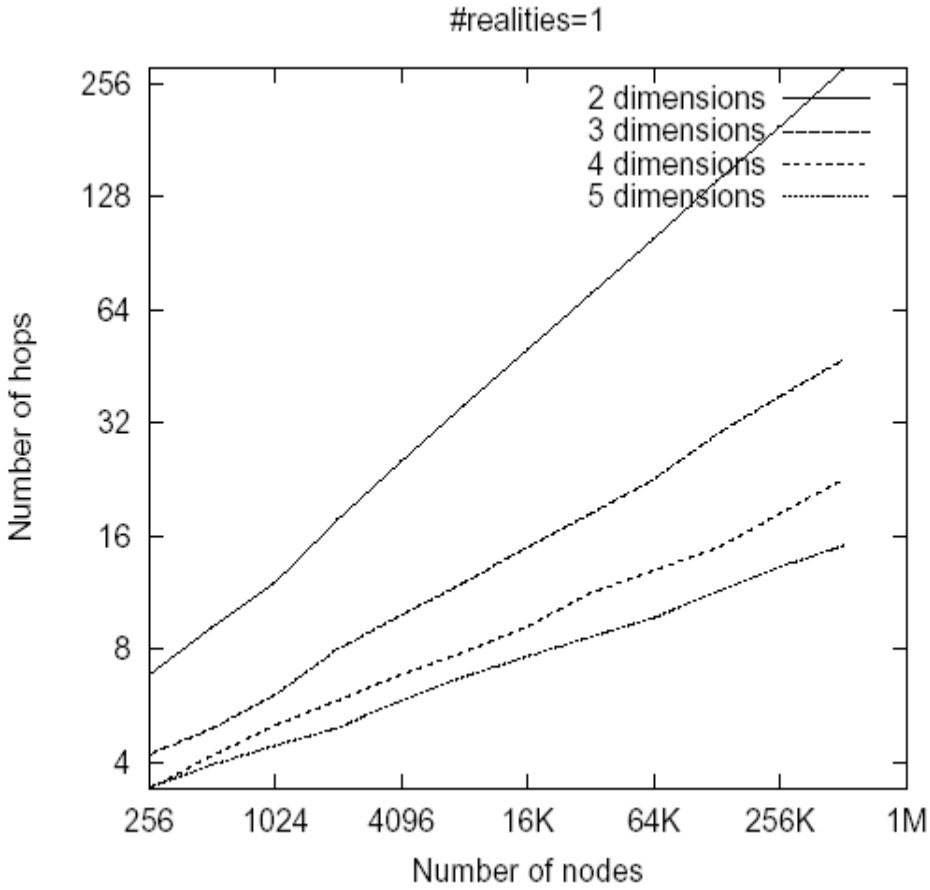
- Graceful departure: node hands over its zone and the (K,V) pairs to a neighbor
- Network failure: unreachable node(s) trigger an immediate takeover algorithm that allocate failed node's zone to a neighbor
  - Detect via lack of periodic refresh messages
  - Neighbor nodes start a takeover timer initialized in proportion to its zone volume
  - Send a TAKEOVER message containing zone volume to all of failed node's neighbors
  - If received TAKEOVER volume is smaller kill timer, if not reply with a TAKEOVER message
  - Nodes agree on neighbor with smallest volume that is alive

CAN Improvements

- CAN provides tradeoff between per-node state, $O(d)$, and path length, $O(dn^{1/d})$
  - Path length is measured in application level hops
  - Neighbor nodes may be geographically distant
- Want to achieve a lookup latency that is comparable to underlying IP path latency
  - Several optimizations to reduce lookup latency also improve robustness in terms of routing and data availability
- Approach: reduce the path length, reduce the per-hop latency, and add load balancing
- Simulated CAN design on Transit-Stub (TS) topologies using the GT-ITM topology generator (Zegura, et. al.)

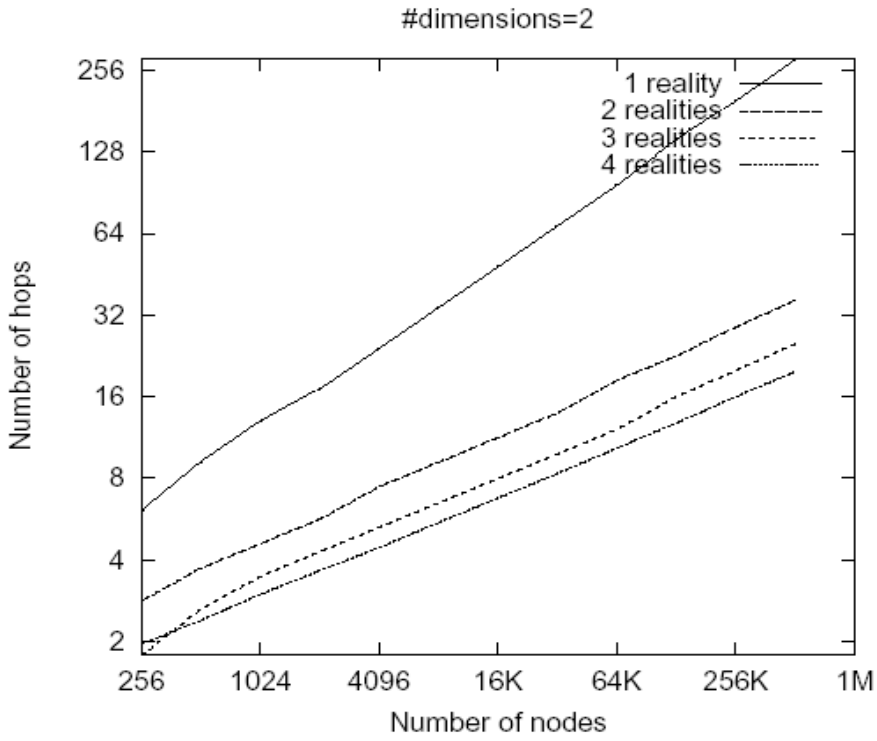
Adding Dimensions

- Increasing the dimensions of the coordinate space reduces the routing path length (and latency)
  - Small increase in the size of the routing table at each node
- Increase in number of neighbors improves routing fault-tolerance
  - More potential next hop nodes
- Simulated path lengths follow $O(dn^{1/d})$



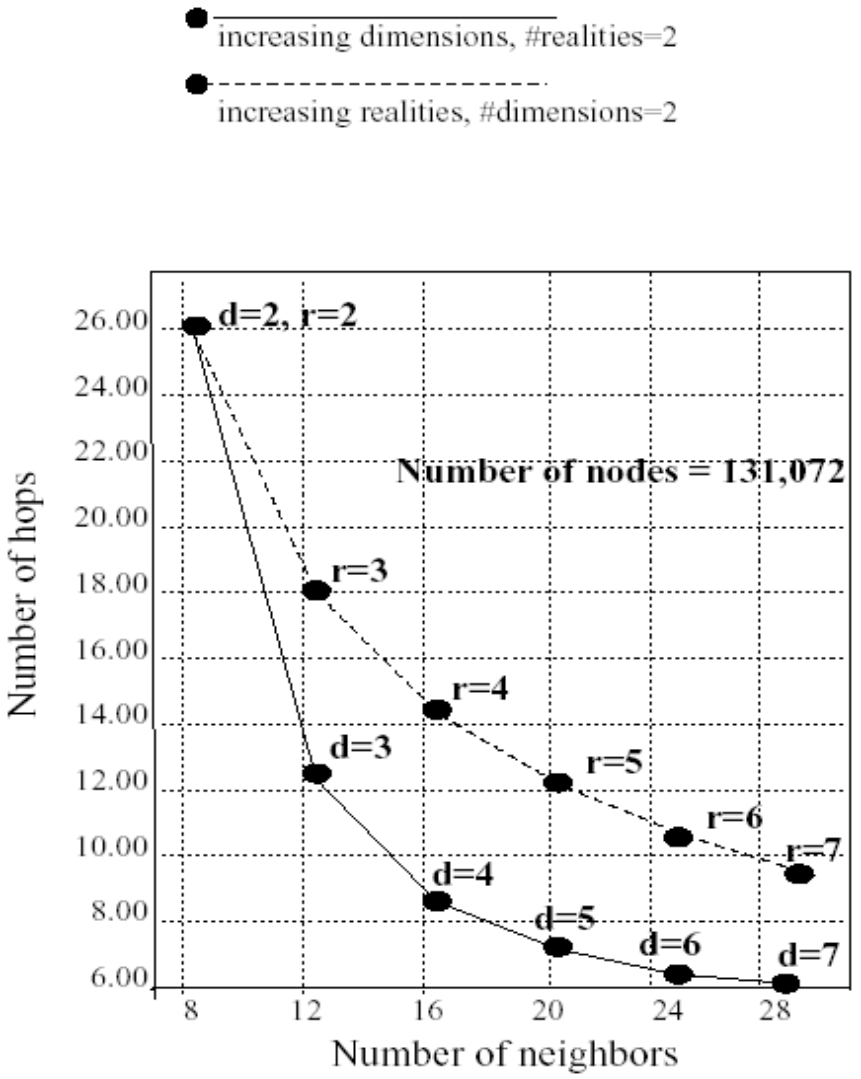
Adding Realities

- Nodes can maintain multiple independent coordinate spaces (realities)
- For a CAN with *r* realities:
  a single node is assigned *r* zones and holds *r* independent neighbor sets
  - Contents of the hash table are replicated for each reality
- Example: for three realities, a (*K*,*V*) mapping to *P*:(*x*,*y*,*z*) may be stored at three different nodes
  - (*K*,*V*) is only unavailable when all three copies are unavailable
  - Route using the neighbor on the reality closest to (*x*,*y*,*z*)



Dimensions vs. Realities

- Increasing the number of dimensions and/or realities decreases path length and increases per-node state
- More dimensions has greater effect on path length
- More realities provides stronger fault-tolerance and increased data availability
- Authors do not quantify the different storage requirements
  - More realities requires replicating (*K*,*V*) pairs



RTT Ratio

- Incorporate RTT in routing metric
  - Each node measures RTT to each neighbor
  - Forward messages to neighbor with maximum ratio of progress to RTT

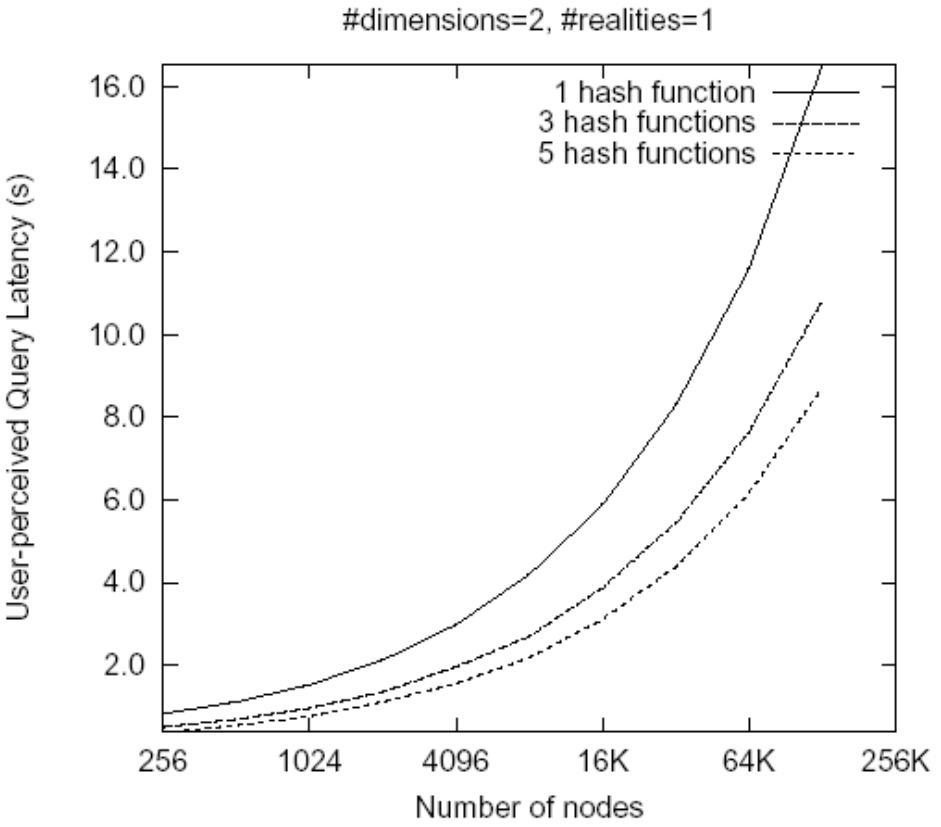
Number of dimensions	Non-*RTT* weighted routing (ms)	*RTT* weighted routing (ms)
2	116.8	88.3
3	116.7	76.1
4	115.8	71.2
5	115.4	70.9

Zone Overloading

- Overload coordinate zones
  - Allow multiple nodes to share the same zone, bounded by a threshold MAXPEERS
  - Nodes maintain peer state, but not additional neighbor state
  - Periodically poll neighbor for its list of peers, measure RTT to each peer, retain lowest RTT node as neighbor
  - (*K*,*V*) pairs may be divided among peer nodes or replicated

Multiple Hash Functions

- Improve data availability by using *k* hash functions to map a single key to *k* points in the coordinate space
- Replicate (*K*,*V*) and store at *k* distinct nodes
- (K,V) is only unavailable when all k replicas are simultaneously unavailable
- Authors suggest querying all k nodes in parallel to reduce average lookup latency



Other optimizations

- Run a background load-balancing technique to offload from densely populated bins to sparsely populated bins (partitions of the space)
- Volume balancing for more uniform partitioning
  - When a JOIN is received, examine zone volume and neighbor zone volumes
  - Split zone with largest volume
  - Results in 90% of nodes of equal volume
- Caching and replication for “hot spot” management

Chord

System Model

- Load balance:
  - Chord acts as a distributed hash function, spreading keys evenly over the nodes.
- Decentralization:
  - Chord is fully distributed: no node is more important than any other.
- Scalability:
  - The cost of a Chord lookup grows as the log of the number of nodes, so even very large systems are feasible.
- Availability:
  - Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, ensuring that, the node responsible for a key can always be found.
- Flexible naming:
  - Chord places no constraints on the structure of the keys it looks up.

System Model

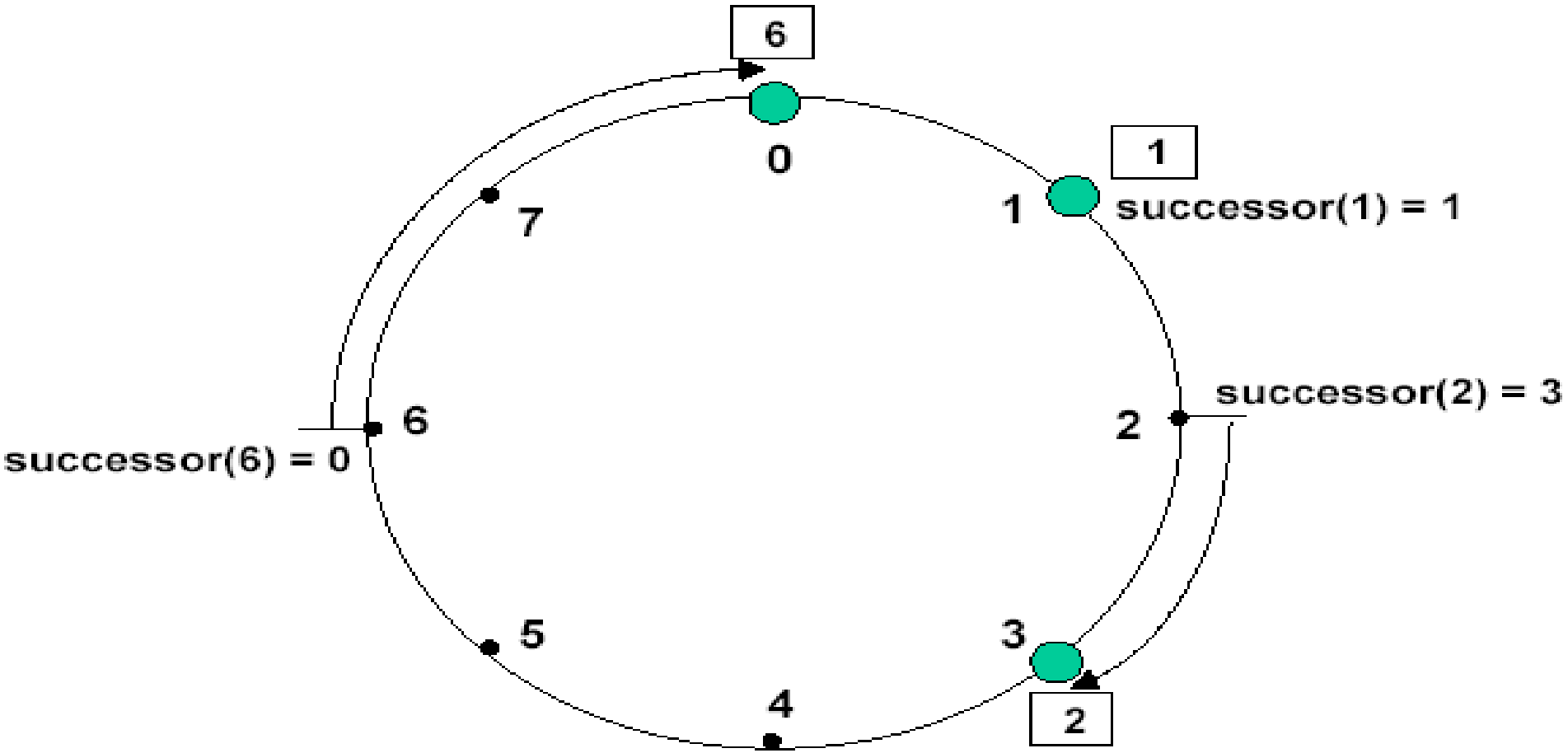
- The application interacts with Chord in two main ways:
  - Chord provides a `lookup(key)` algorithm that yields the IP address of the node responsible for the key.
  - The Chord software on each node notifies the application of changes in the set of keys that the node is responsible for.

The Chord Protocol

- The Chord protocol specifies how to find the locations of keys.
- It uses consistent hashing, all nodes receive roughly the same number of keys.
- When an N^{th} node joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location.
- In an N-node network, each node maintains information only about $O(\log N)$ other nodes, and a lookup requires $O(\log N)$ messages.

Consistent Hashing

- The consistent hash function assigns 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.
- Key *k* is assigned to the first node whose identifier is equal to or follows *k* in the identifier space. This node is called the *successor node* of key *k*.
- If identifiers are represented as a cycle of numbers from 0 to $2^m - 1$, then *successor*(*k*) is the first node clockwise from *k*.



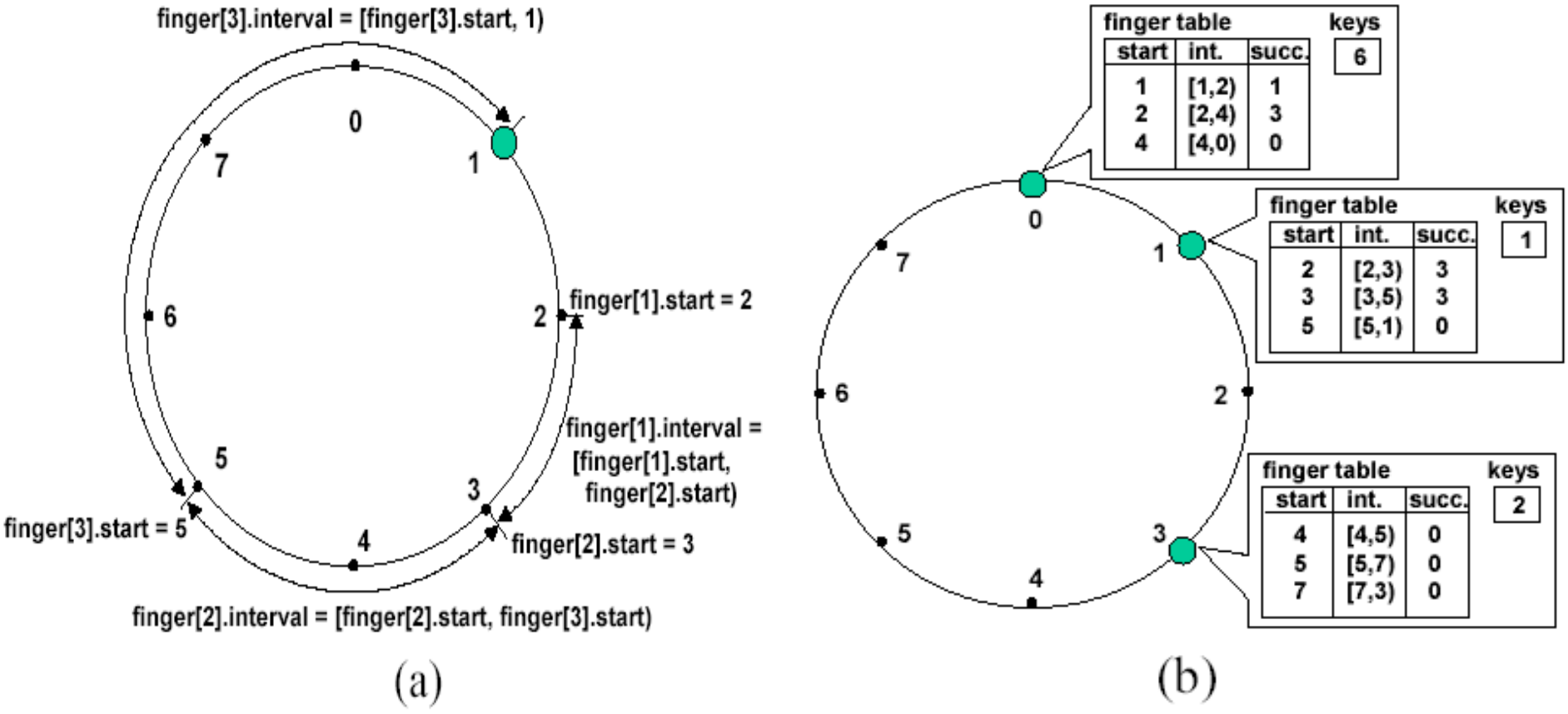
An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.

Scalable key Location

- Let *m* be the number of bits in the key/node identifiers.
- Each node, *n*, maintains a routing table with (at most) *m* entries, called the *finger table*.
- The i^{th} entry in the table at node *n* contains the identity of the *first* node, *s*, that succeeds *n* by at least $2^i - 1$ on the identity circle.

Notation	Definition
finger[*k*].*start*	$(n + 2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
.*interval*	[*finger*[*k*].*start*, *finger*[*k* + 1].*start*)
.*node*	first node $\geq n$.*finger*[*k*].*start*
successor	the next node on the identifier circle; *finger*[1].*node*
predecessor	the previous node on the identifier circle

Definition of variables for node *n*, using *m*-bit identifiers.



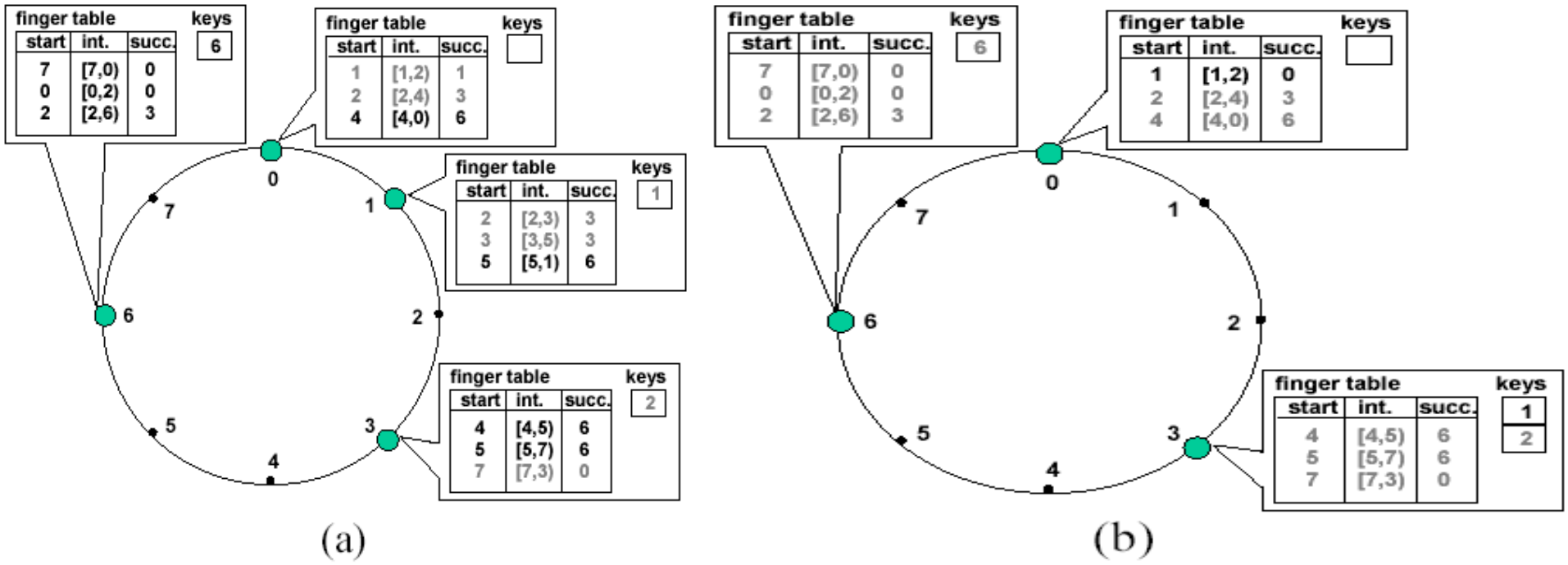
(a) The finger intervals associated with node 1. (b) Finger tables and key locations for a net with nodes 0, 1, and 3, and keys 1, 2, and 6.

Scalable key Location

- With high probability (or under standard hardness assumption), the number of nodes that must be contacted find a successor in an *N*-node network is *O* (log *N*).

Node Joins

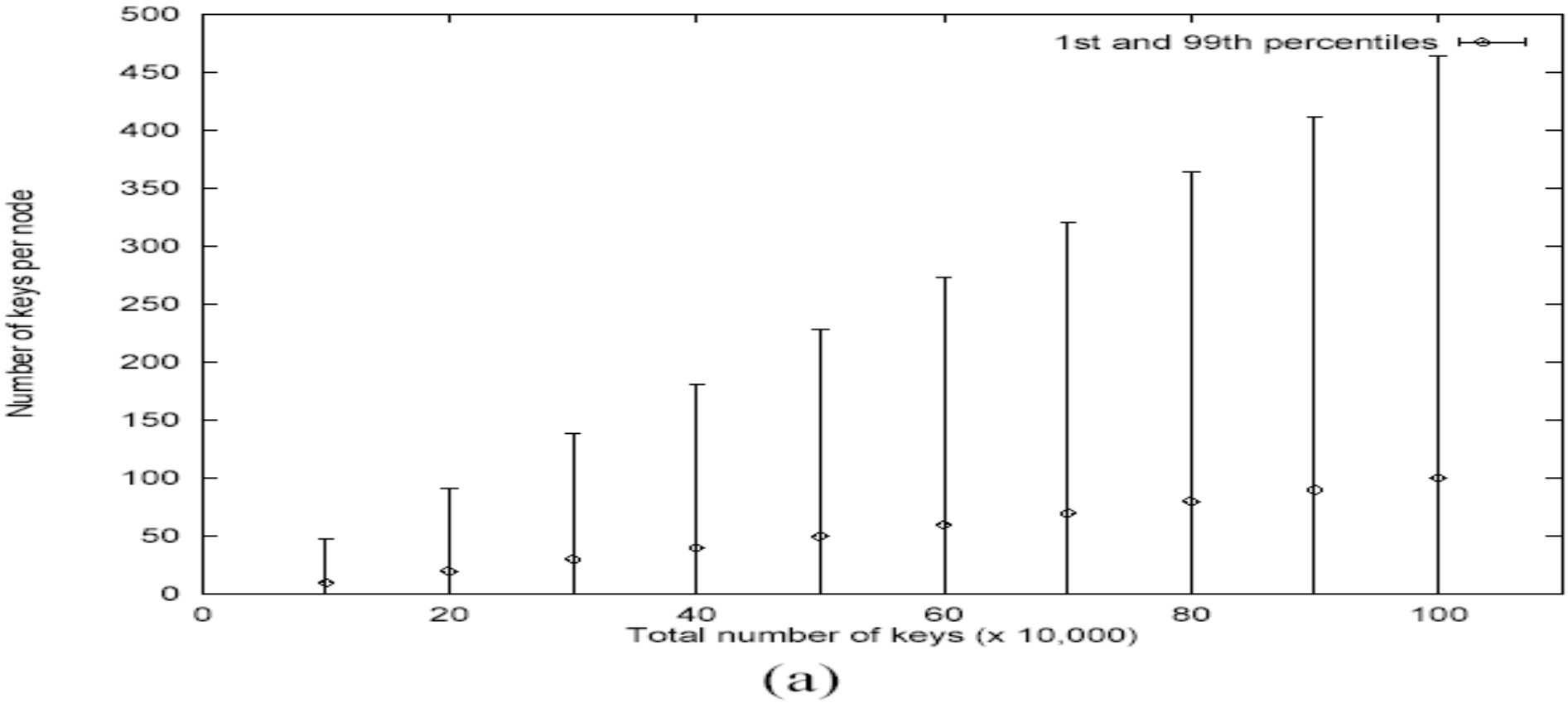
- Each node in Chord maintains a *predecessor pointer*, and can be used work counterclockwise around the identifier circle.
- When a node *n* joins the network:
  - Initialize the predecessor and fingers of node *n*.
  - Update the fingers and predecessors of existing nodes to reflect the addition of *n*.
  - Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node *n* is now responsible for.



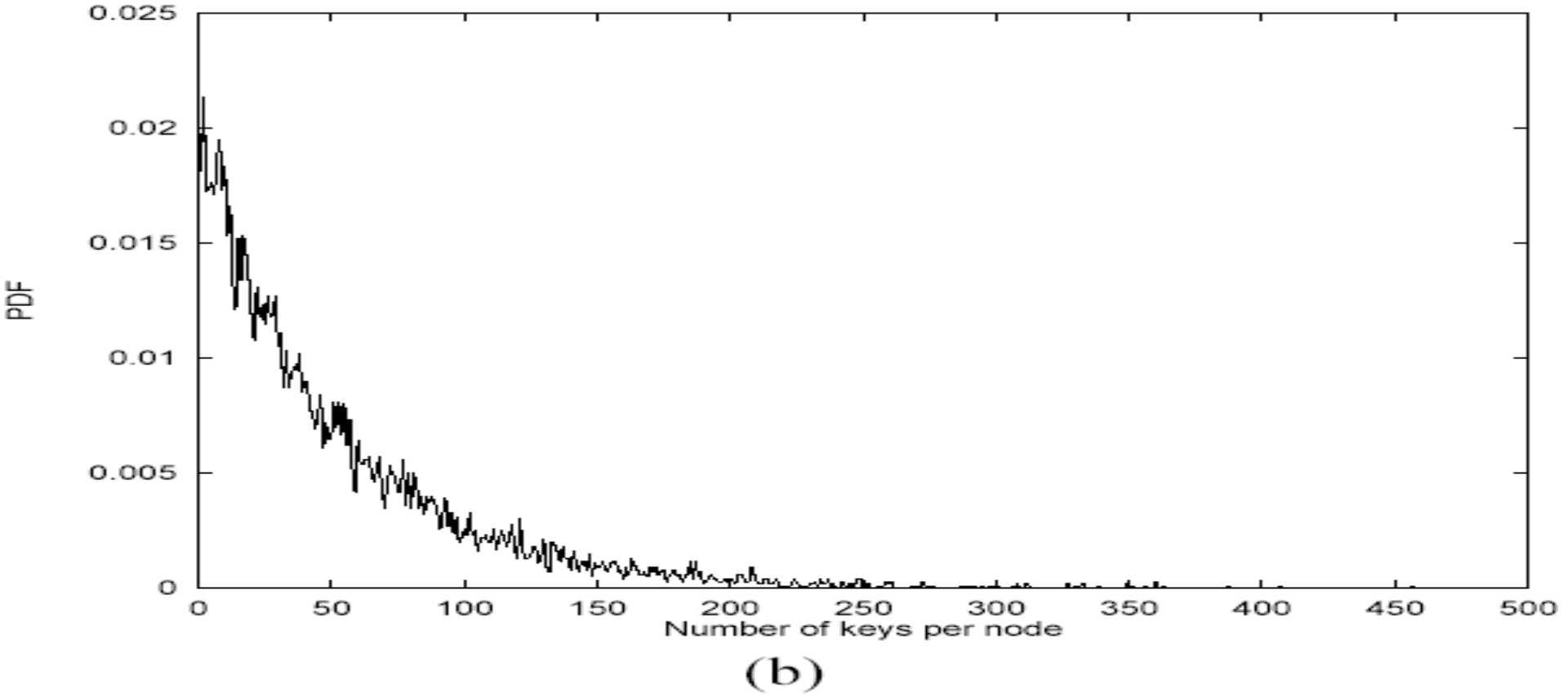
(a) Finger tables and key locations after node 6 joins. (b) Finger table and key locations after node 1 leaves. Changed entries are shown in black , and unchanged in gray.

Failures and Replication

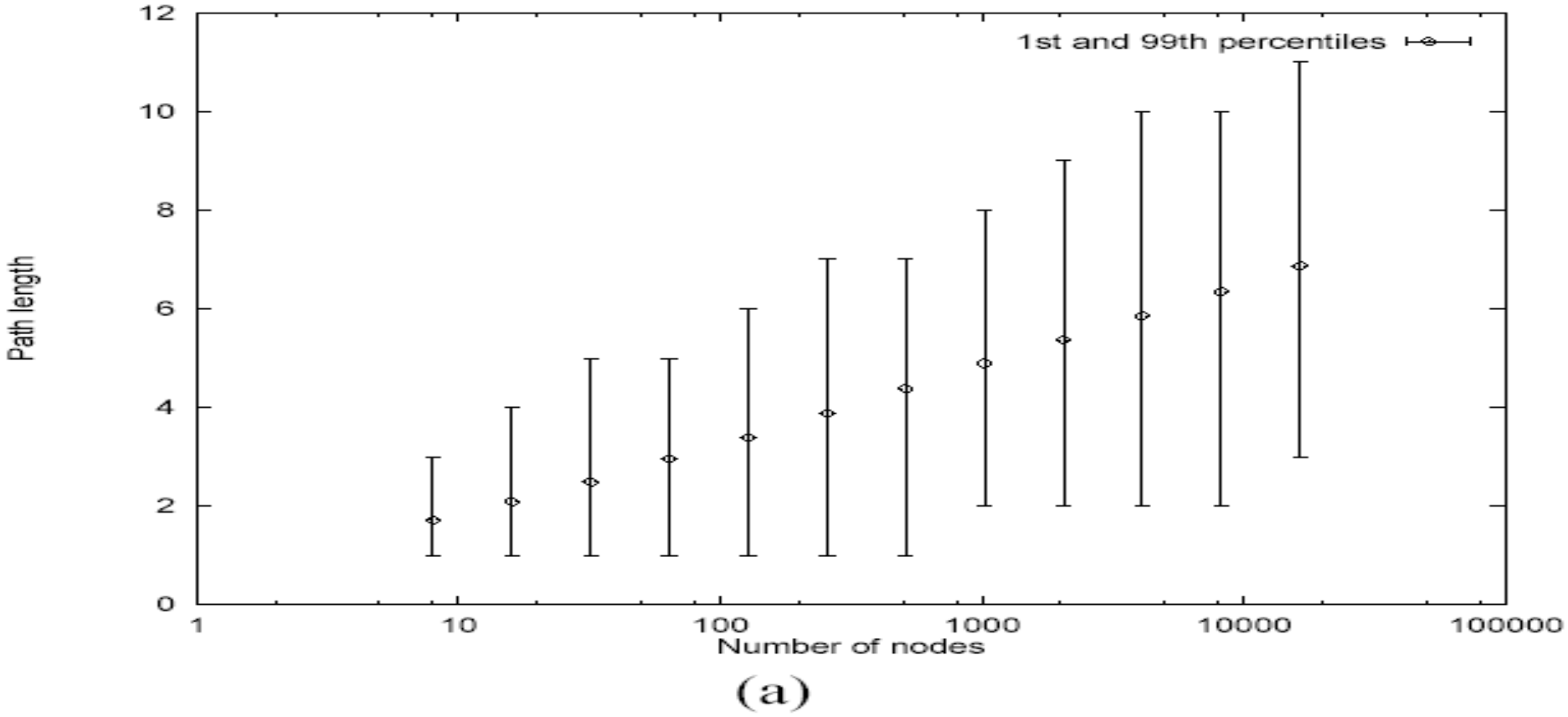
- When a node *n* fails, nodes whose finger tables include *n* must find *n*'s successor.
- Each Chord node maintains a "successor-list" of its r nearest successor on the Chord ring.
- A typical application using Chord might store replicas of the data associated with key at the *k* nodes succeeding the key.



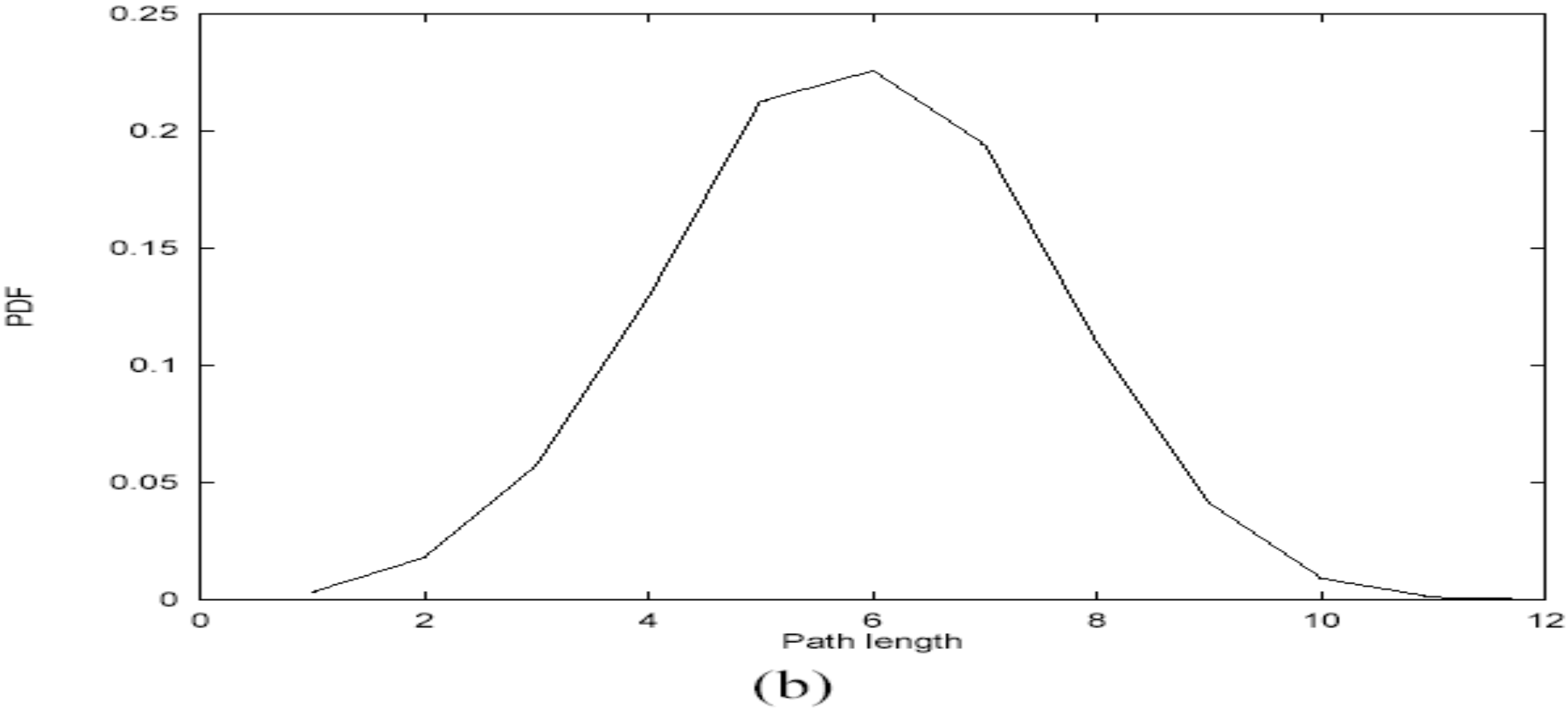
The mean and 1st and 99th percentiles of the number of keys stored per node in a 10^4 node network.



The probability density function (PDF) of the number of keys per node. The total number of keys is 5×10^5.



The path length as a function of network size.



The PDF of the path length in the case of a 2^{12} node network.

BATON
A Balanced Tree Structure for Peer-to-Peer Networks

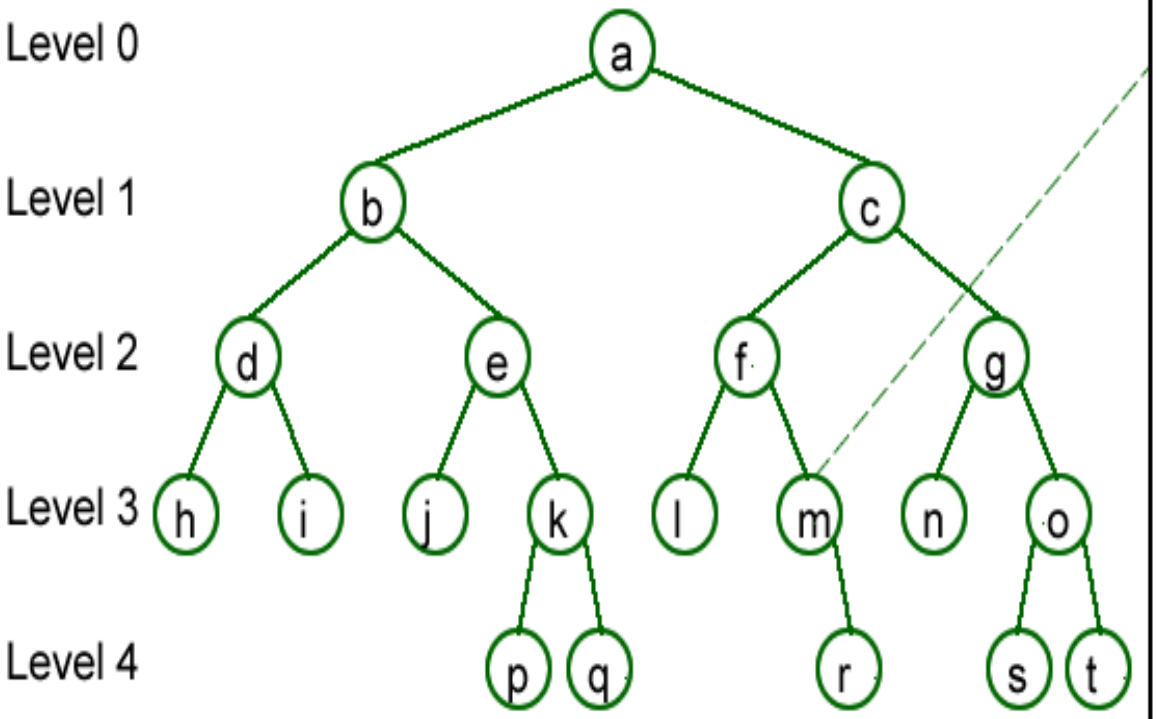
H. V. Jagadish, Beng Chin Ooi, Quang Hieu Vu

Related Work

- P-Grid (CoopIS'01)
  - Based on a binary prefix tree structure.
  - ↳ Can not guarantee *log N search step* boundary
- P-Tree (WebDB'04)
  - Based on B+-tree structure and uses CHORD as overlay framework
  - Each node maintains a branch of the B+tree
  - ↳ *Expensive cost* to keep consistence knowledge among nodes
- Multi-way tree (DBISP2P'04)
  - Each node maintains links to its parent, its siblings, its neighbors, and its children
  - ↳ Can not guarantee *log N search step* boundary

BATON Architecture

- BATON: BAlanced Tree Overlay Network
- *Definition: A tree is balanced if and only if at any node in the*



Node m: level=3, number=6
parent=f, leftchild=null, rightchild=r
leftadjacent=f, rightadjacent=r

Left routing table:

	Node	Left child	Right child	Lower bound	Upper bound
0	l	null	null	l_{lower}	l_{upper}
1	k	p	q	k_{lower}	k_{upper}
2	i	null	null	i_{lower}	i_{upper}

Right routing table:

	Node	Left child	Right child	Lower bound	Upper bound
0	n	null	null	n_{lower}	n_{upper}
1	o	s	t	o_{lower}	o_{upper}

Binary Balanced Tree Index Architecture

Theorems

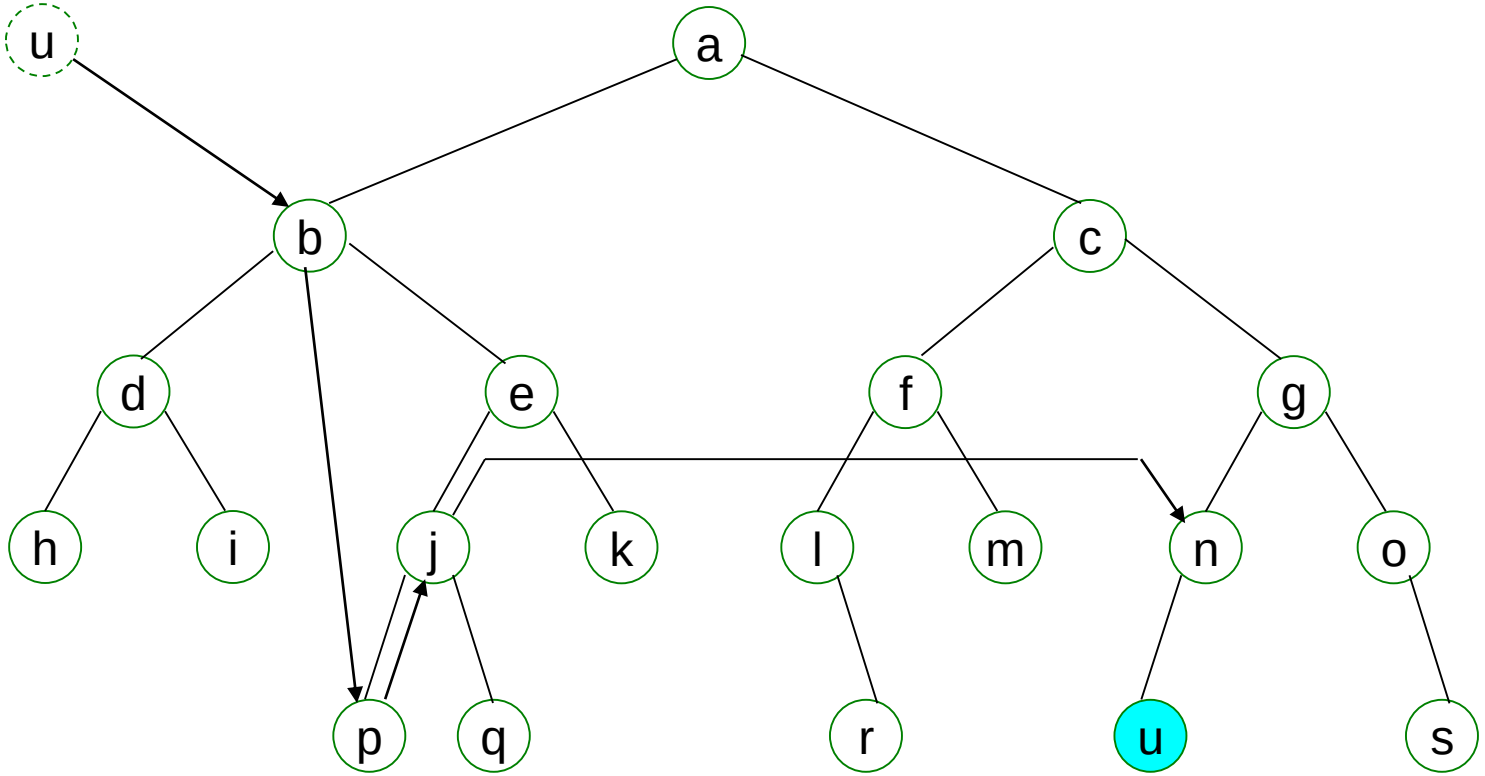
- *Theorem 1: The tree is a balanced tree if every node in the tree that has a child also has both its left and right routing tables full (*).*

- *Theorem 2: If a node, say x, contains a link to another node, say y, in its left or right routing tables, parent node of x must also contain a link to parent node of y unless the same node is parent of both x and y.*

(*) A routing table is full if none of the valid links is NULL.

Node join

- Example: new node *u* joins the network



Node join

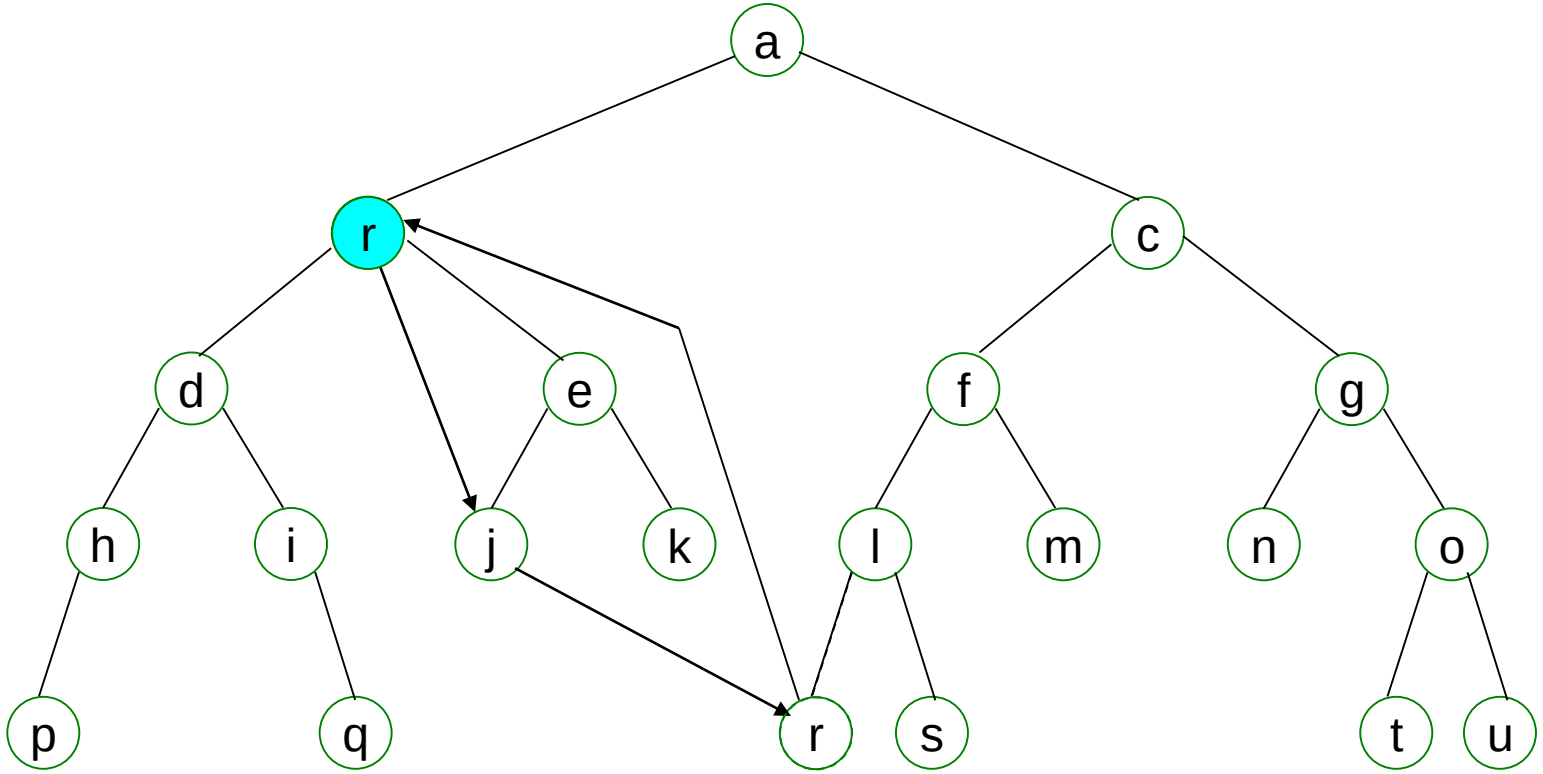
- Cost of finding a node to join: O(log N)
- When a node accepts a new node as its child
  - Split half of its content (its range of values) to its new child
  - Update adjacent links of itself and its new child
  - Notify both its neighbor nodes and its new child's neighbor nodes to update their knowledge
  - Cost: 6 log N

Node departure

- When a node wishes to leave the network
  - If it is a leaf node and there is no neighbor node having children, it can leave the network
    - Transfer its content to the parent node, and update correspondence adjacent link
    - Notify its neighbor nodes and its parent's neighbor nodes to update their knowledge
    - Cost: 4 log N
  - If it is a leaf node and there is a neighbor node having children, it needs to find a leaf node to replace it by sending a FINDREPLACEMENTNODE request to a child of that neighbor node
  - If it is an intermediate node, it needs to find a leaf node to replace it by sending a FINDREPLACEMENTNODE to one of its ***adjacent*** nodes

Node departure

- Example: existing node *b* leaves the network



Node departure

- Cost of finding a leaf node to replace: O(log N)
- When a node comes to replace a leave node
  - Notify its parent and its neighbor nodes as in case of leaf node leaving: 4 log N
  - Notify its new parent node, its new neighbor nodes, and the parent's neighbor nodes: 4 log N
  - Total cost: 8 log N

Fault tolerance

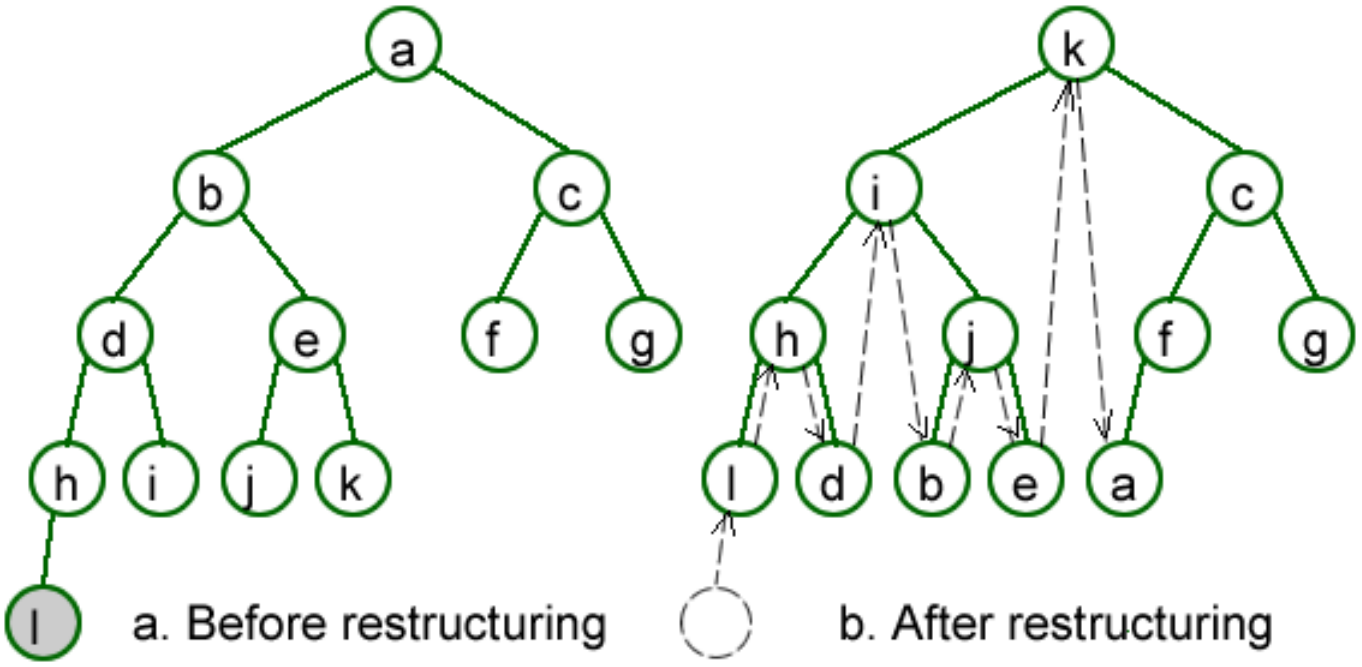
- Node failure
  - Nodes discovering failure of a node report to that node's parent.
  - The failure's parent node will take responsibility for finding a leaf node to replace if necessary.
  - Routing information of the failure node can be recovered by contacting its neighbor nodes via routing information of its parent.
- Fault tolerance: failure node can be passed by two ways
  - Through routing tables (similar to CHORD) - ***horizontal axis***
  - Through parent-child and adjacent links - ***vertical axis***
  - Specifically, even if all nodes at the same level fail, the tree is not partitioned

Network restructuring

- Necessary in case of forced join or forced leave that is used in load balancing scheme
- Network restructuring is triggered when the condition in the theorem 1 is violated
- Network restructuring is done by shifting nodes via adjacent links
  - No data movement is required
  - Each shifted node requires O(log N) effort to update routing tables

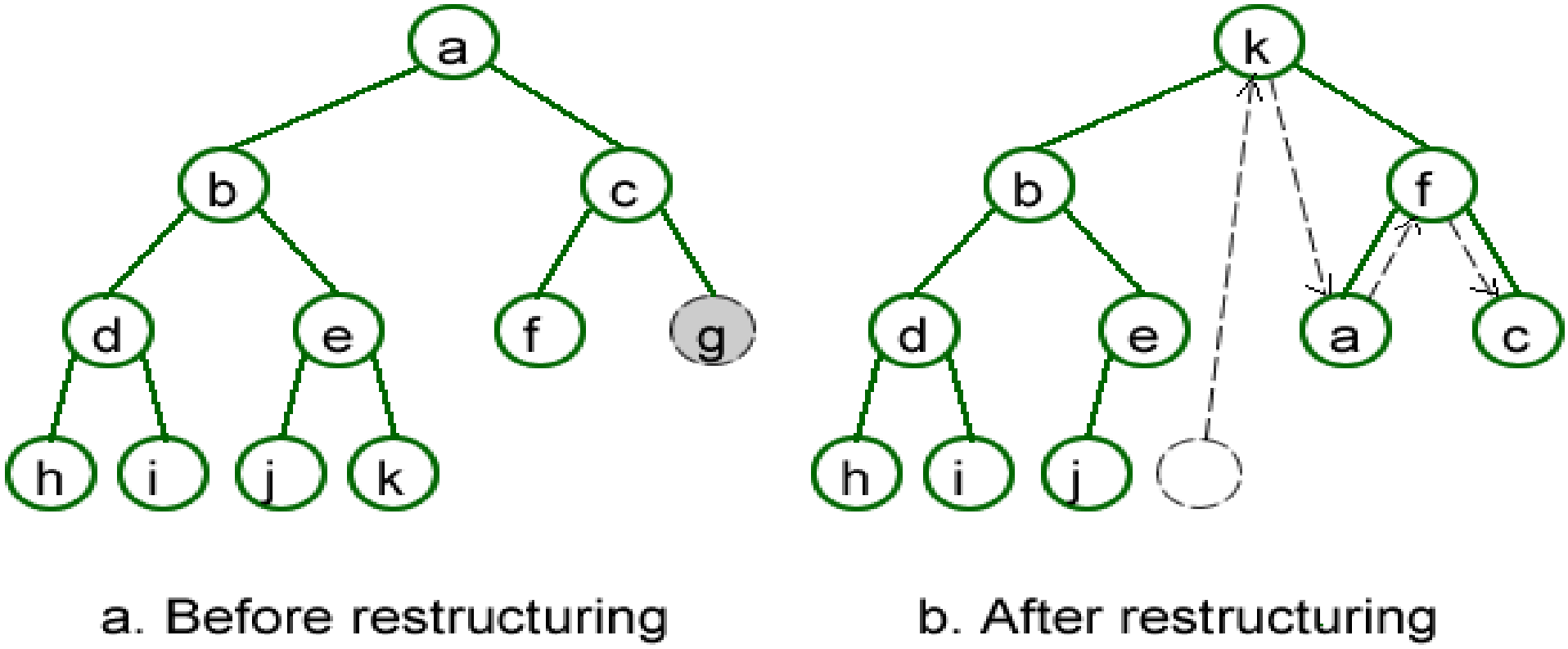
Forced join

- Example 1: network restructure is triggered as a forced join



Forced leave

- Example 2: network restructuring is triggered as a forced leave

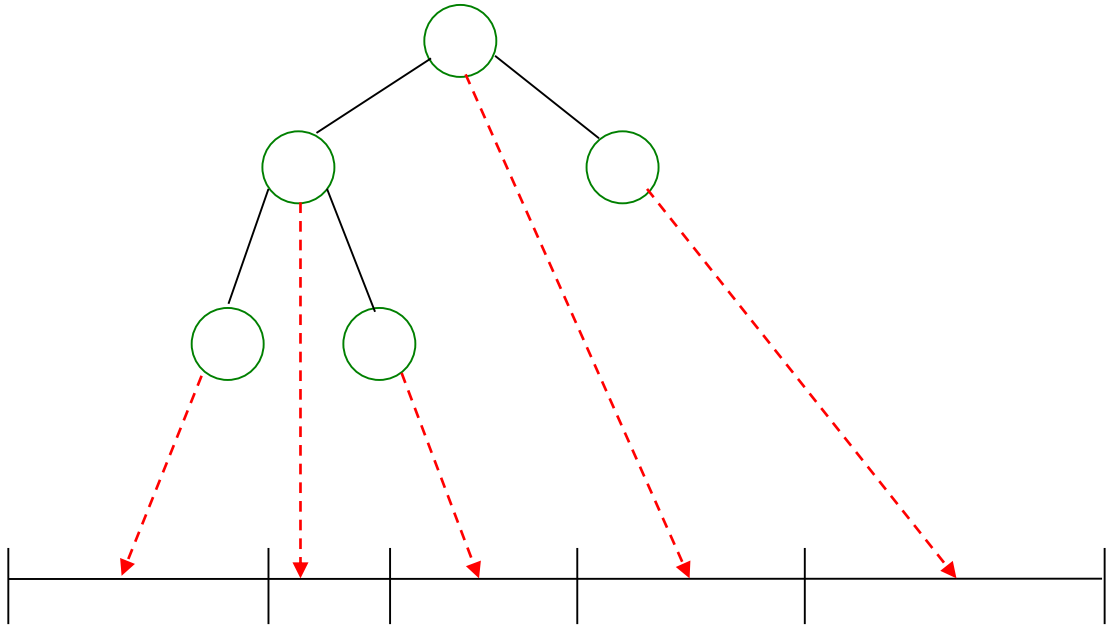


a. Before restructuring

b. After restructuring

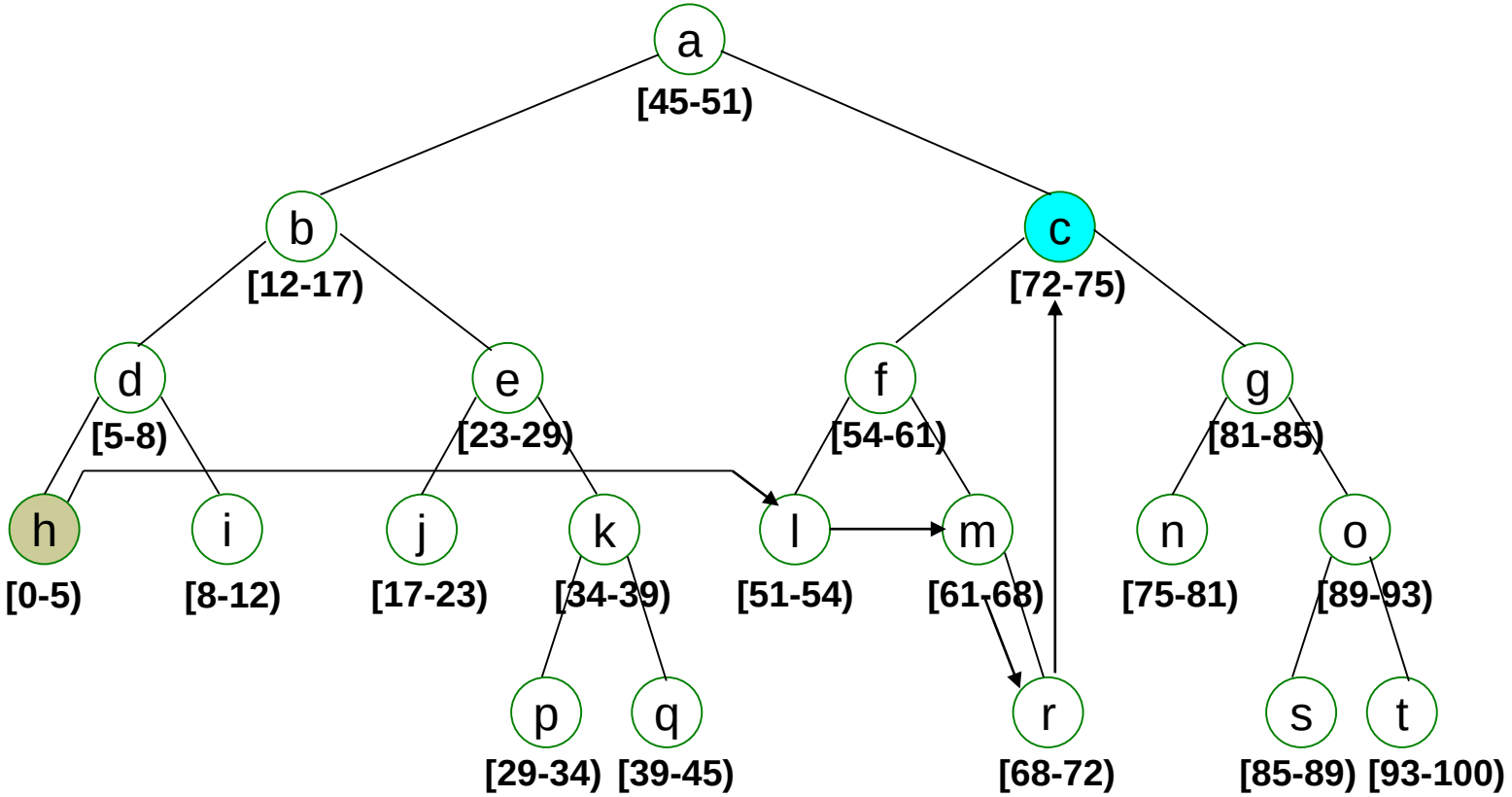
Index construction

- Each node is assigned a range of values
- The range of values directly managed by a node is
  - Greater than the range managed by its left adjacent node
  - Smaller than the range managed by its right adjacent node



Exact match query

- Example node wants to search data belonged to node *c*, say **74**



Range query

- Process similar to exact match query
  - First, find an intersection with searched range
  - Second, follow adjacent links to retrieve all results
- Cost
  - Exact match query: O(log N)
  - Range query: O(log N + X) where X is the total number of nodes containing searched results

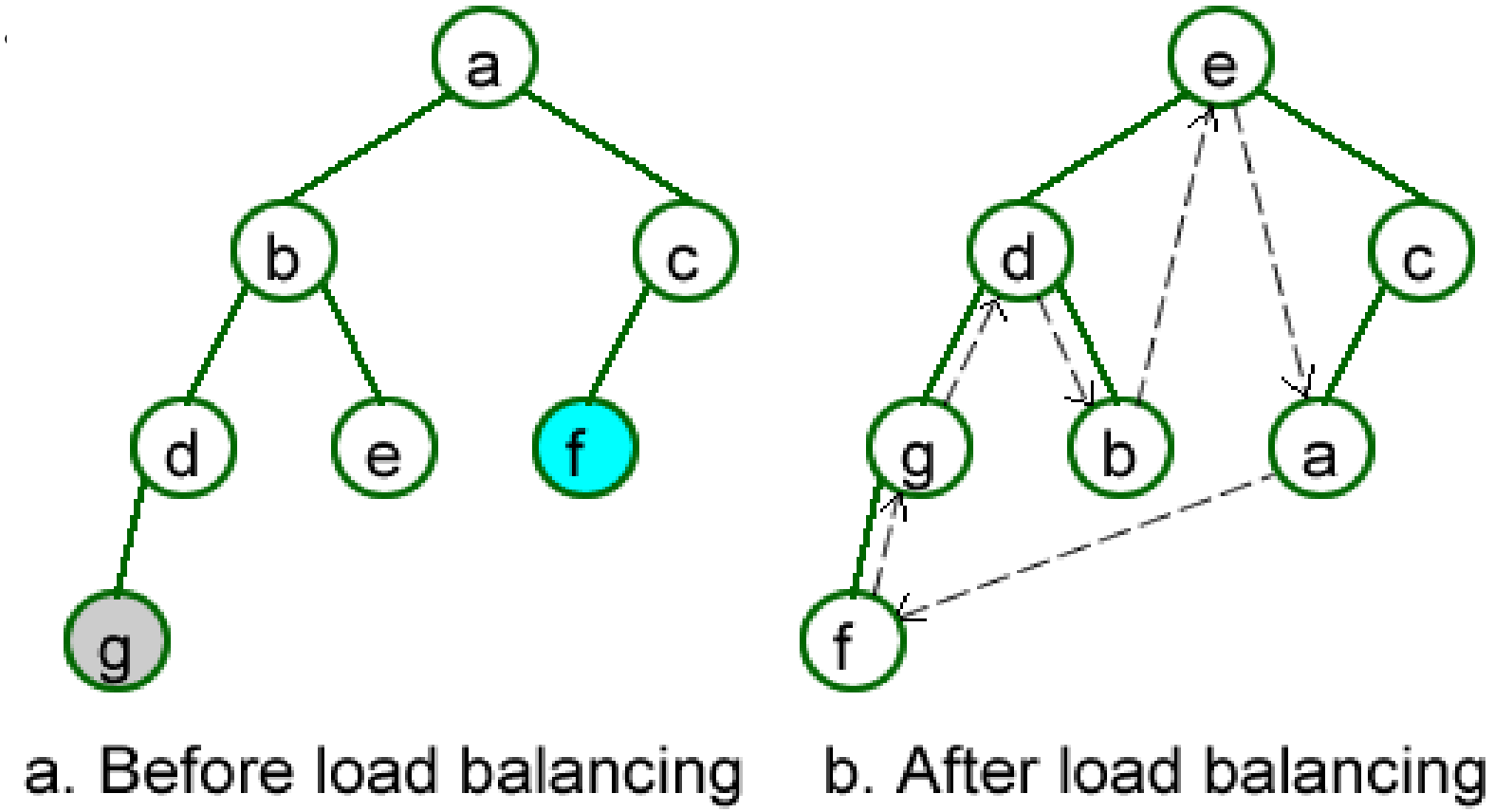
Data insertion and deletion

- Insertion
  - Follow the exact match query process to find the node where data should be inserted except that
    - If it is the left most node and the inserted value is still less than its lower bound, or if it is the right most node and the inserted value is still greater than its upper bound, it expands its range of values to accept the new inserted value.
    - In this case, additional log N cost is needed for updating routing tables
- Deletion
  - Follow the exact match query process to find the node containing data which should be deleted
- Cost: similar to exact match query process
  - O(log N) for both insertion and deletion

Load balancing

- Load balancing process is initialized when a node is overloaded or under loaded due to insertion or deletion
- 2 load balancing schemes
  - Do load balancing with adjacent nodes
  - An overloaded node finds a lightly loaded node to share work load (only if the overloaded / under loaded node is a leaf node)
    - A lightly loaded node is found by traveling through neighbor nodes within O(logN) steps.
    - Once found, the lightly loaded node transfers its content to one of its adjacent nodes, forced leaves its current position, and forced joins as a child of the overloaded node.
    - Network restructuring is triggered if necessary
  - Similar process is applied to under loaded nodes
- Cost: O(log N) for each node attending load balancing process

Load balancing

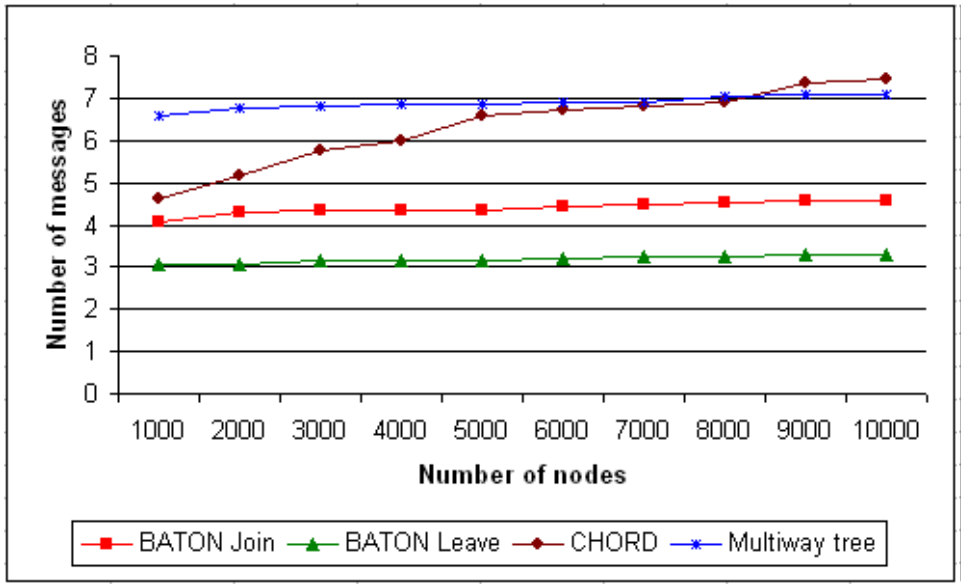


a. Before load balancing b. After load balancing

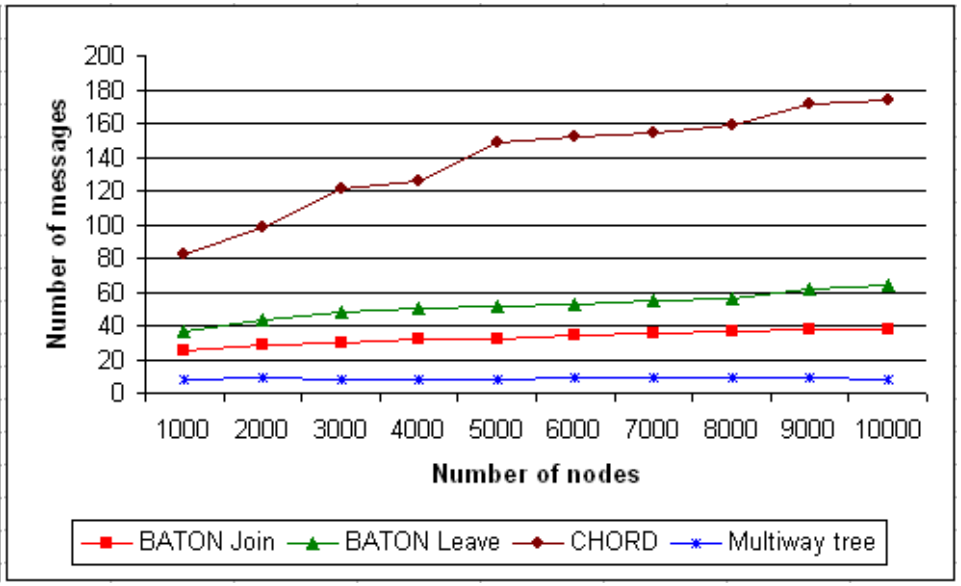
Experimental study

- Experimental setup
  - Test the network with different number of nodes N from 1000 to 10000.
  - For a network of size N, 1000 x N data values in the domain of [1, 1000000000) are inserted in batches
  - 1000 exact queries, and 1000 range queries are executed
  - CHORD and Multi-way tree are used to compare

Join and leave operations

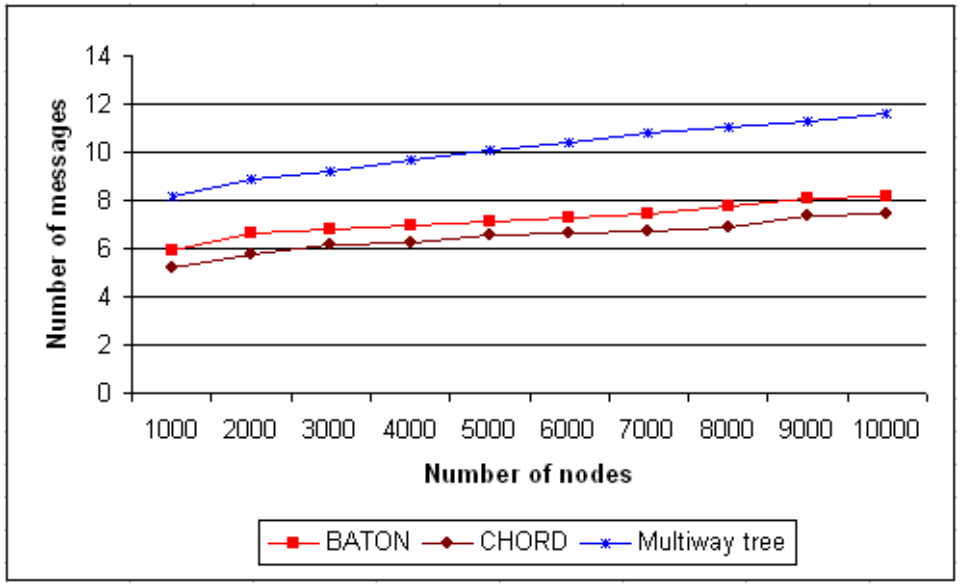


Cost of finding join node and replacement node



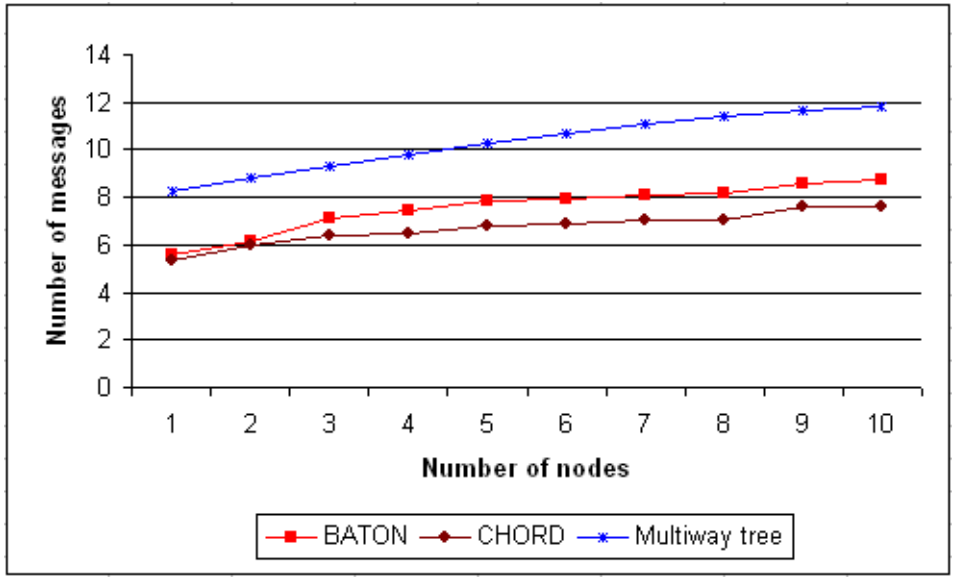
Cost of updating routing tables

Insert and delete operations

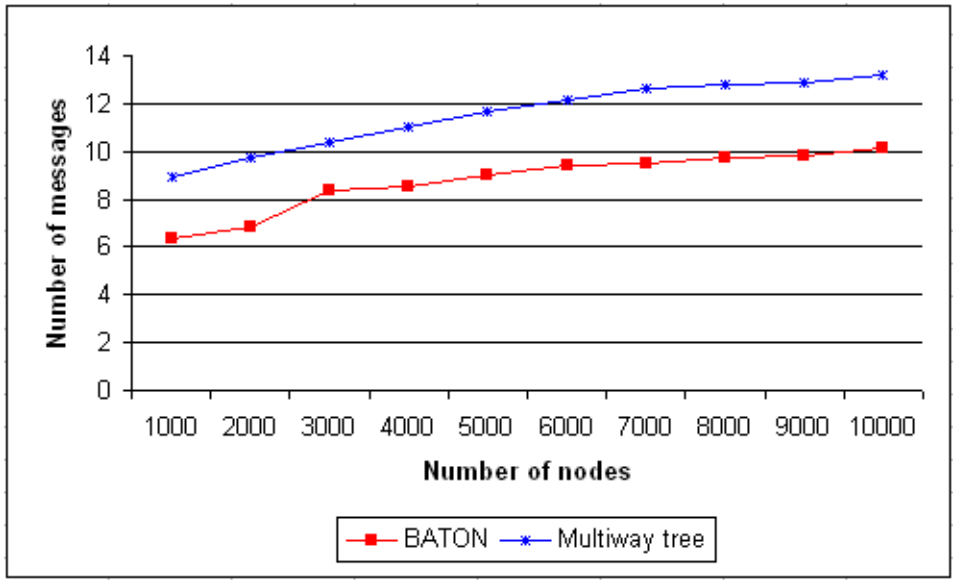


Cost of insert and delete operations

Search operations

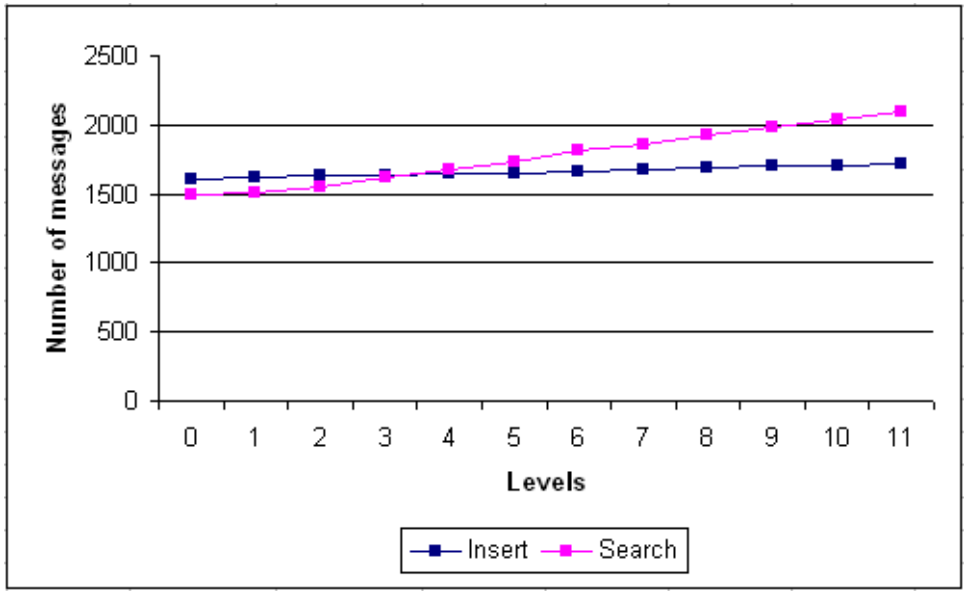


Cost of exact match query



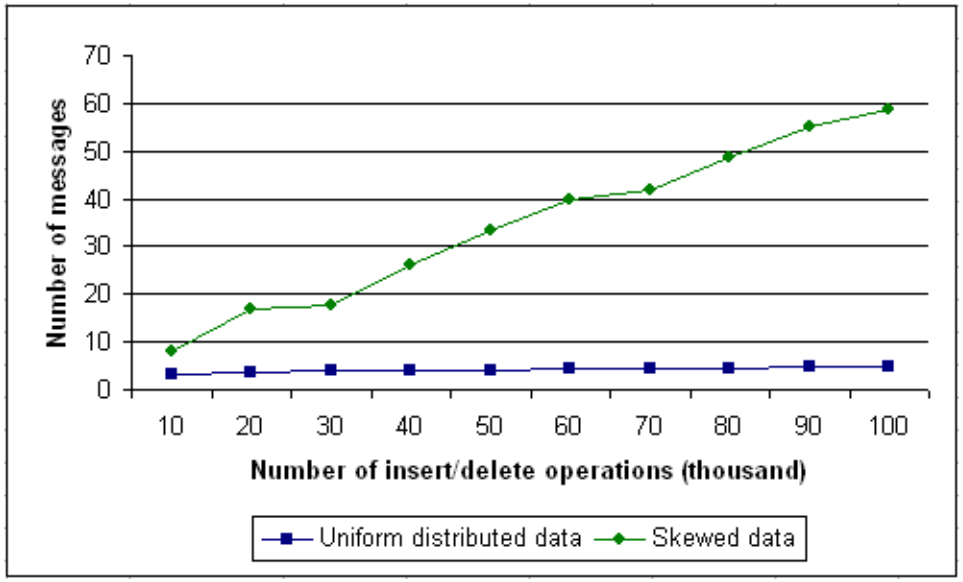
Cost of range query

Access load

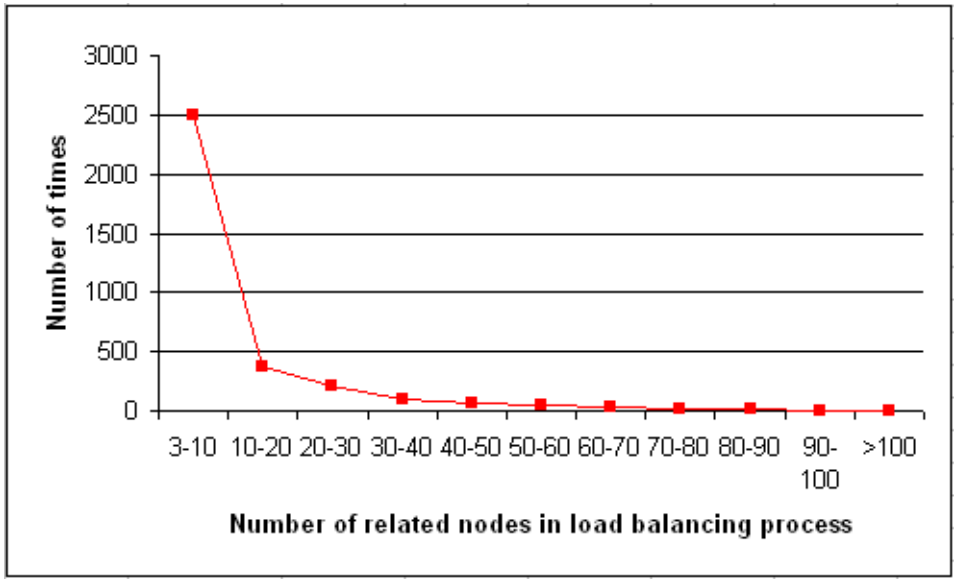


Access load for nodes at different levels

Effect of load balancing

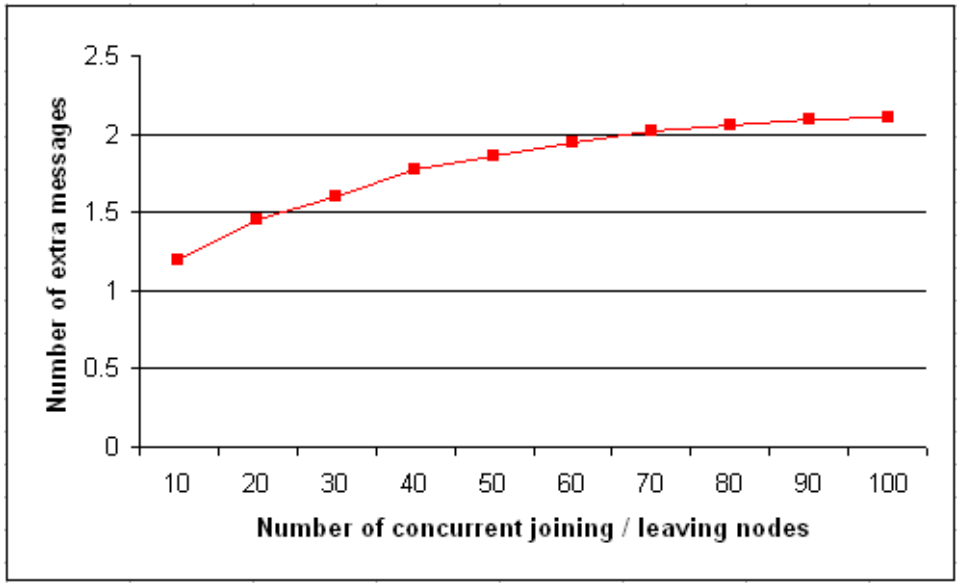


Average messages of load balancing operation



Size of load balancing process

Effect of network dynamics



Network Dynamics

Conclusion

- BATON
  - The first P2P overlay network based on a balanced tree structure
  - Strengths
    - Incur less cost of updating routing tables compared to other systems
    - Support both exact match query and range query efficiently
    - Flexible and efficient load balancing scheme
    - Scalability (NOT bounded by network size or ID space before hand)

Thank you
Q & A