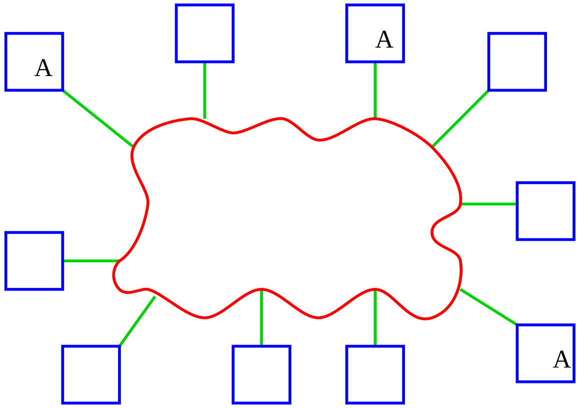
# On Tracking Distributed Objects

**Rajmohan Rajaraman**

**Northeastern University**

# The Data Tracking Problem




- A data tracking scheme.
  - Find a (nearby) copy of the requested object.
  - Insert/delete object copies.
  - Update control information as nodes join/leave the system.
- Basic problem in distributed systems [Mullender-Vitányi 88, Awerbuch-Peleg 90, Guyton-Schwarz 95, . . . ].

# Data Tracking Operations

- *find*$(u, x)$: Issued by node $u$ to locate a copy of object $x$.
- *insert*$(u, x)$: Node $u$ inserts a new copy of object $x$.
- *delete*$(u, x)$: Node $u$ deletes an existing copy of object $x$.
- *join*$(u)$: Node $u$ joins the system.
- *leave*$(u)$: Node $u$ leaves the system.

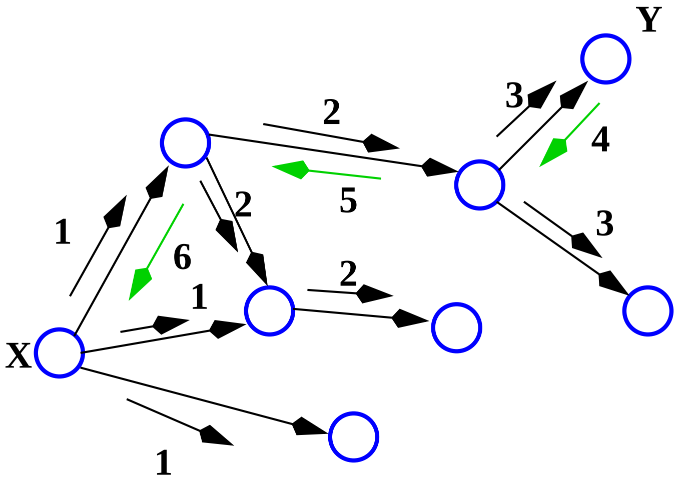
# Applications

- DNS:
  - *find* maps names to IP addresses.
- Peer-to-peer networks:
  - Each node is a client and a server.
  - Need to provide efficient operations with lightweight nodes.
- Replicated servers.
  - The *join* and *leave* operations may be ignored.
- Tracking mobile users.
  - No copies.

# Current Methods in P2P File-Sharing

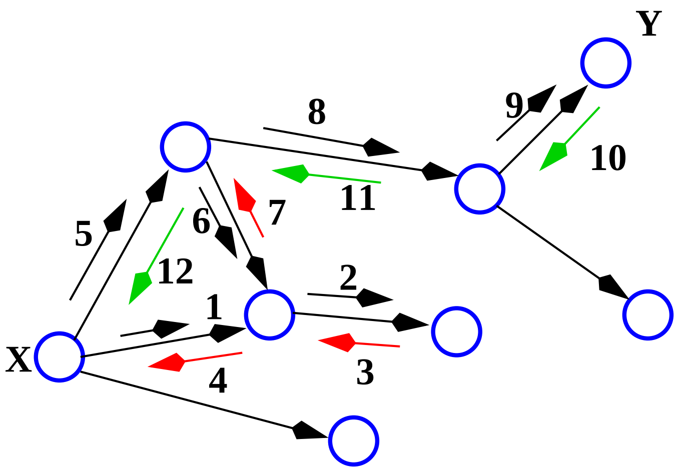
- Popular commercial systems:
  - Napster
  - Gnutella
  - Freenet [Clarke et al 00]
- Selected academic research projects:
  - Oceanstore [Kubiatowicz et al 00]
  - Chord [Stoica et al 01]
  - Content Addressable Network (CAN) [Ratnasamy et al 01]

# Gnutella




- Controlled flooding.
- Efficient in terms of *find* cost: each request is satisfied by a nearby copy.
- Not scalable: in the worst-case, the entire network may be flooded.
  - Susceptible to denial-of-service attacks.
  - A time-to-live (TTL) field eliminates loops and may prevent excessive flooding.

# Freenet




- "Sequential version" of flooding.
- Trades off efficiency for scalability.
  - Little congestion is caused due to a single request.
  - Inefficient: A request may have to be forwarded along a long chain of nodes before being satisfied.
  - Need to query neighbors in order of "likelihood" of holding the object.

## Measures

- Communication cost of *find*, *insert*, and *delete* operations.
  - *cost* is an idealized function of latency, bandwidth, queue sizes, etc.
  - For analysis, assume a static cost; also often assume that it is a metric.
  - Let $v$ be the "nearest node" to $u$ that has a copy of $x$.

$$\text{Stretch of } \mathit{find}(u,x) = \frac{\text{Cost of } \mathit{find}(u,x)}{\mathit{cost}(u,v)}.$$

- Join and leave operations:
  - Communication cost incurred.
  - Number of nodes that are updated.

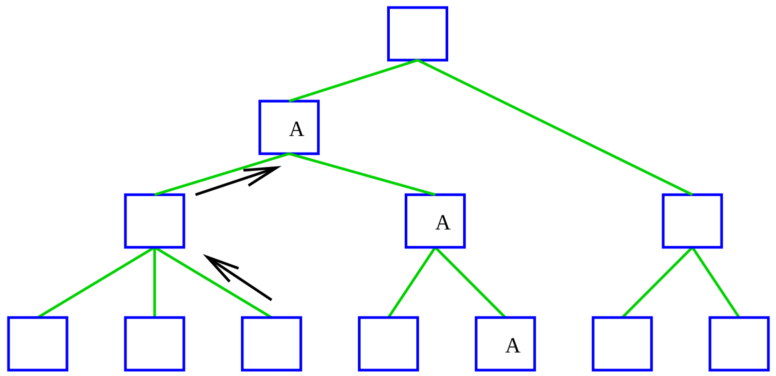
# Measures, contd.

- Memory overhead: The maximum amount of control information stored at a node of the network.
  - List of nodes that the node forwards requests to.
  - List of objects that the node is aware of.
- Load at a node:
  - Static load: number of objects it is aware of.
  - Dynamic load: number of find operations affecting the node per unit time.

# Outline of Ideas

- Sparse neighborhood covers [Awerbuch-Peleg 90]:
  - Addresses locality.
  - One can prove near-optimal bounds on stretch factor.
  - Resultant network decomposition has many potential applications.
  - Somewhat complicated and may be hard to update when nodes leave/join.
- A simpler flat tracking scheme [Plaxton et al 97]:
  - Partially addresses locality.
  - Addresses static load balancing.
- Consistent hashing and variants [Chord, CAN]:
  - Adaptive to node joins/leaves.
  - Addresses static load balancing.

# A Tree-Based Distributed Solution




- Embed a tree into the network:
  - The embedding must respect network locality.
  - The tree and its embedding determine the location of control information among the network nodes.
  - Forward the request up the tree until a copy is located, e.g., in DNS.

# Embedding Trees into Arbitrary Metrics

- Easy to see that tree embeddings may not preserve locality.
- Embed multiple "tree-like structures":
  - Sparse neighborhood covers [Awerbuch-Peleg 90].
  - Hierarchically well-separated trees [Bartal 96].

# Sparse Neighborhood Covers

- For each node $u$ and cost $c$, define

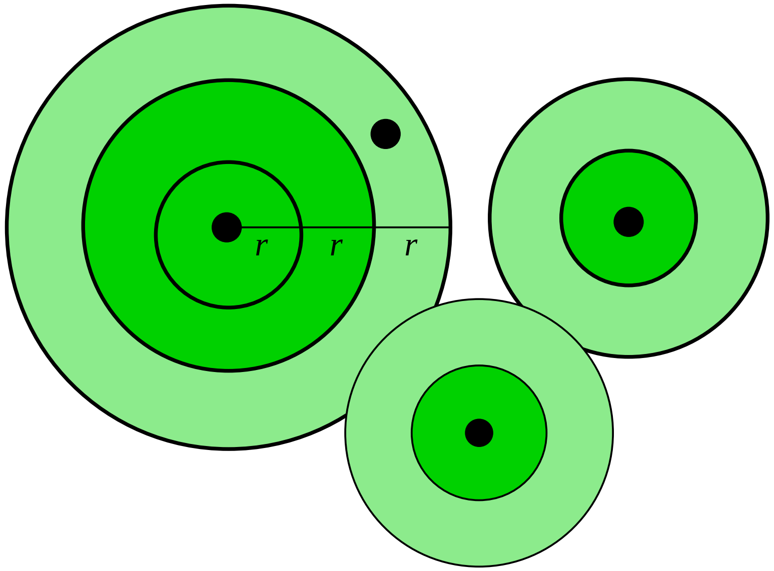
$$N(u, c) = \{v : cost(u, v) \le c\}.$$

- A sparse $2^i$-cover $M_i$ is a collection of sets of nodes (clusters)
  - For each $u$, some $S$ in $M_i$ contains $N(u, 2^i)$.
  - Diameter of each cluster is $O(2^i \log n)$.
  - Each node belongs to $O(\log n)$ clusters.

# Finding Sparse $2^i$-Covers

- Repeat the following until all nodes are "removed".
  - Find smallest $j$ such that $2|N(u, j2^i)| \geq |N(u, (j+1)2^i)|$.
  - Either $j \leq \log n$ exists or $N(u, 2^i \log n)$ includes all nodes; in latter case, set $j = \log n$.
  - Include set $N(u, (j+1)2^i)$ in cover.
  - Mark all nodes in $N(u, (j+1)2^i)$ and "remove" all nodes in $N(u, j2^i)$ from further consideration.
  - Pick an unmarked node $u$ and go to step 1.
  - If no unmarked node, then unmark all nodes and go to step 1.
- When a node $v$ is "removed", $N(v, 2^i)$ is in some cluster.
- Each node is in $O(\log n)$ clusters.

# Sparse $2^i$-Cover Computation



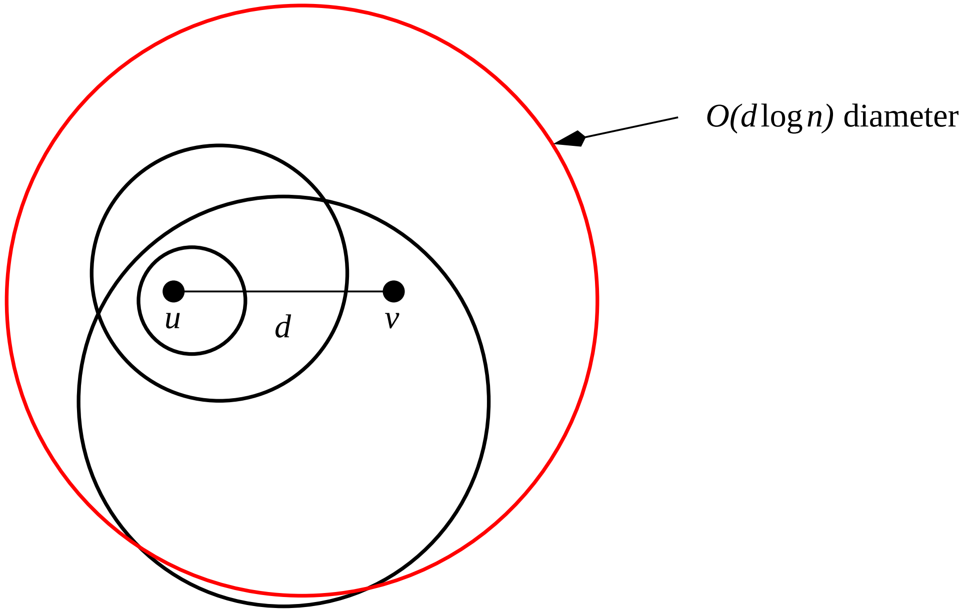

- A distributed randomized algorithm can be used to compute sparse covers [Linial-Saks 91].

- Runs in polylogarithmic time whp.

# Sparse Covers and Data Tracking

- Compute sparse $2^i$-cover for all $i \leq \log(Diam)$.
- Elect a leader in each cluster.
- *find*: For each $i$, node $u$ queries leader of "home cluster" in $2^i$-cover, until object located.
- *insert/delete*: For each $i$, node $u$ informs leader of each cluster containing $u$ in $2^i$-cover.

# Finding an Object



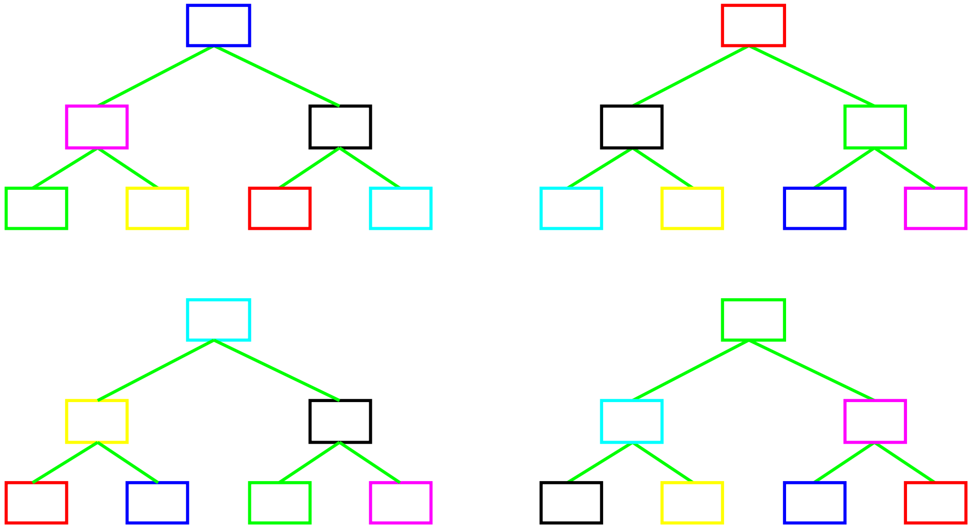

- Cost of find is

$$O\left(\sum_{i=0}^{\lceil \log d \rceil} (d/2^i) \log n\right) = O(d \log n).$$

# Complexity of Measures

- Stretch of find is $O(\log n)$.
- Insert/delete:
  - Worst-case cost is $O(Diam \operatorname{polylog}(n))$
  - Amortized stretch of $O(\operatorname{polylog}(n))$ can be achieved [Bartal-Fiat-Rabani 92].
- Memory overhead: Some “leader” nodes need $\Omega(m)$ storage, where $m$ is the number of objects.
- Join/leave: Requires $\Omega(n)$ nodes to be updated in worst case.

# A Collection of Trees



- For each object, have a logical tree.
- Randomly map the logical tree among the nodes, respecting locality.
- The set of object copies that need to be tracked is evenly distributed.
- Scalability problem: Each node has to know its neighbors in each tree.

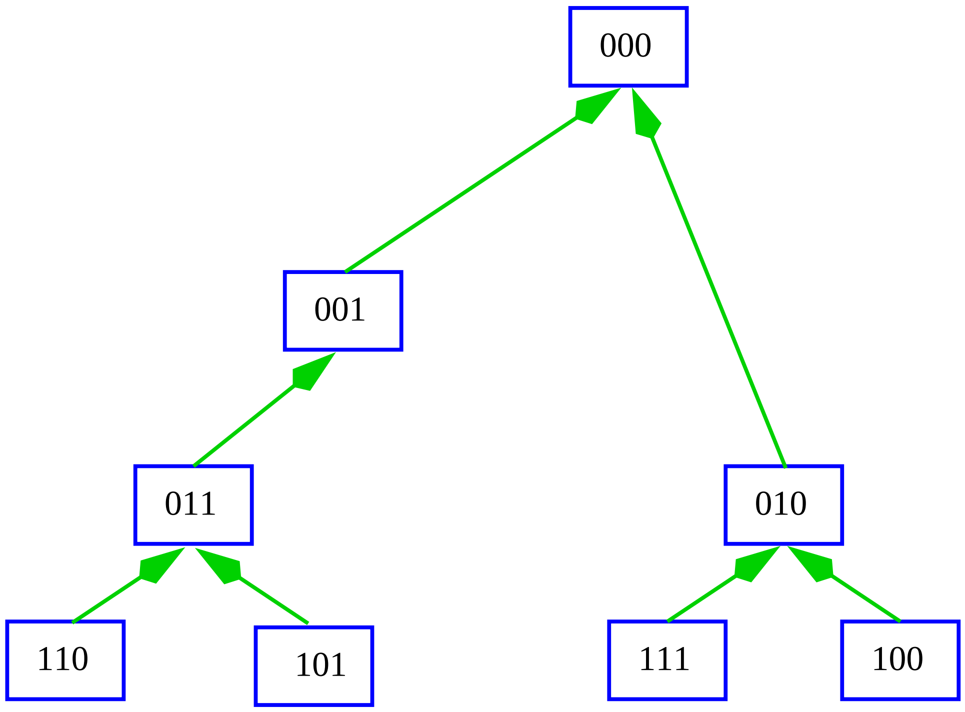
# A Simpler Flat Tracking Scheme

- A randomized embedding of logical trees that achieves (static) load balancing and can be stored with low memory overhead.
- For a restricted class of cost functions, it achieves asymptotically efficient cost.
- Forms the data location component of Oceanstore.

# Object and Node IDs

- Assign unique IDs to objects and nodes.
- Object-location information will be assigned to nodes by matching IDs.
  - For example, the node whose bits match the largest prefix of object ID is a "root" node for the object; it has information about at least one copy of the object.
  - If nodes have random IDs, the tracking scheme is topology-sensitive.

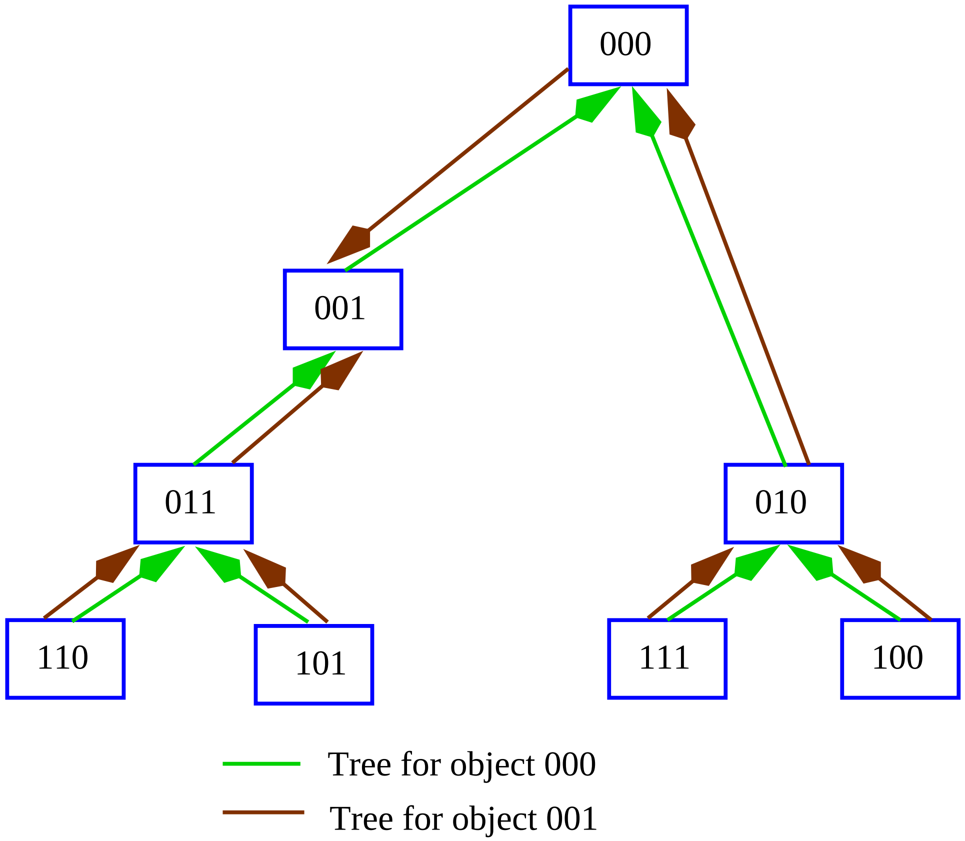
# An Access Tree




Tree for object 000

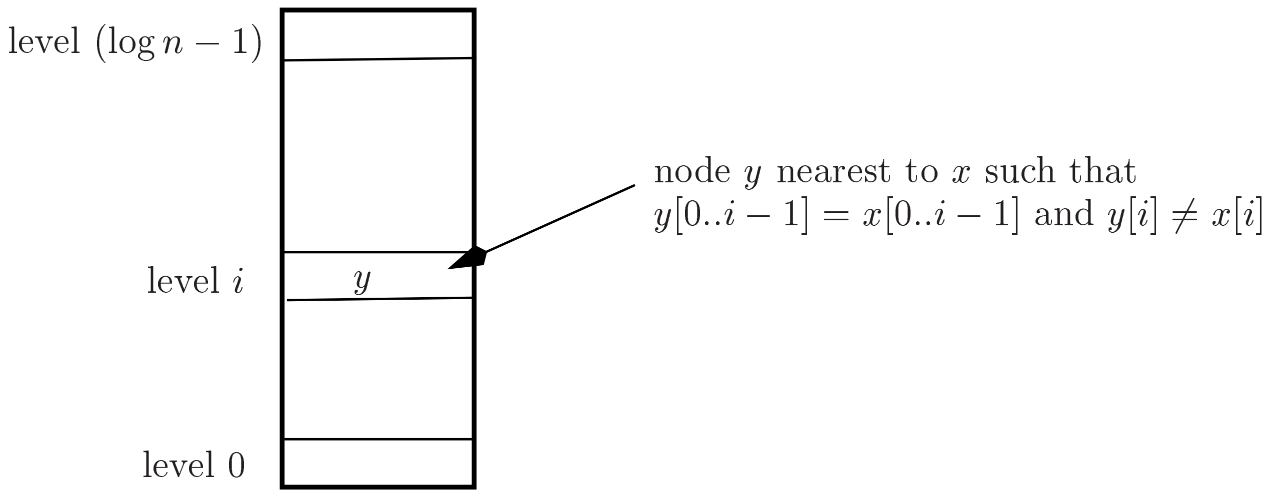
- The parent of a node $u$ is the closest node whose id matches $A$'s id in a longer prefix than $u$'s id.

# Overlapping Access Trees




- The neighbors in different access trees overlap; the degree of any node across all across trees is $\log n$.

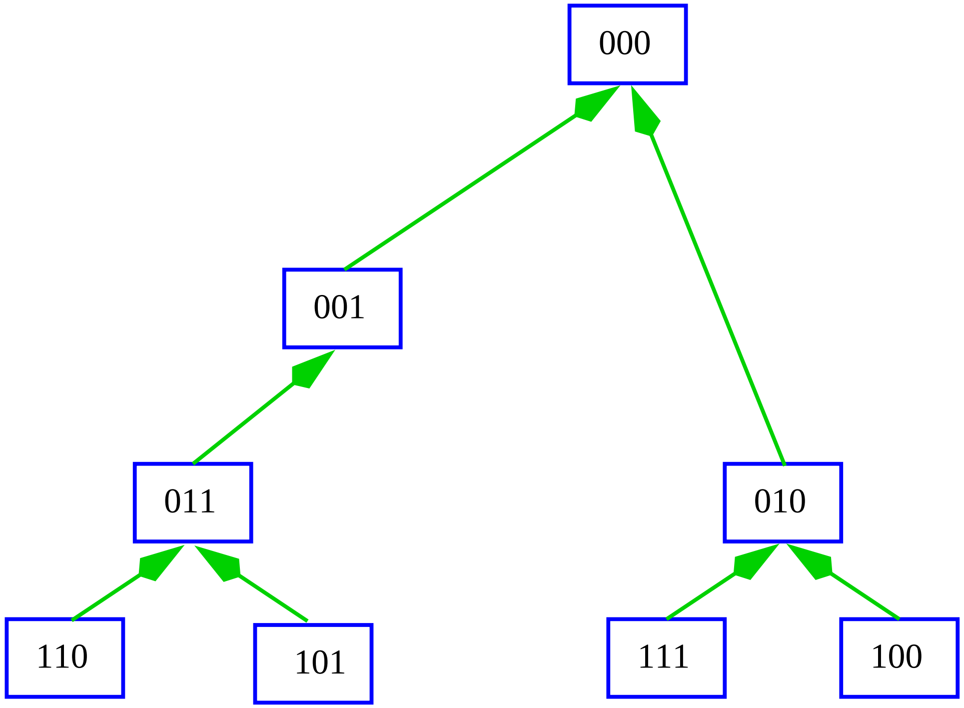
# Neighbor Tables




Neighbor Table for node $x$

- For $0 \leq i < \log n$, the $i$-neighbor of $x$ is the nearest node $y$ such that
  - $y[0..i-1]$ matches $x[0..i-1]$.
  - $y[j]$ is different from $x[j]$.

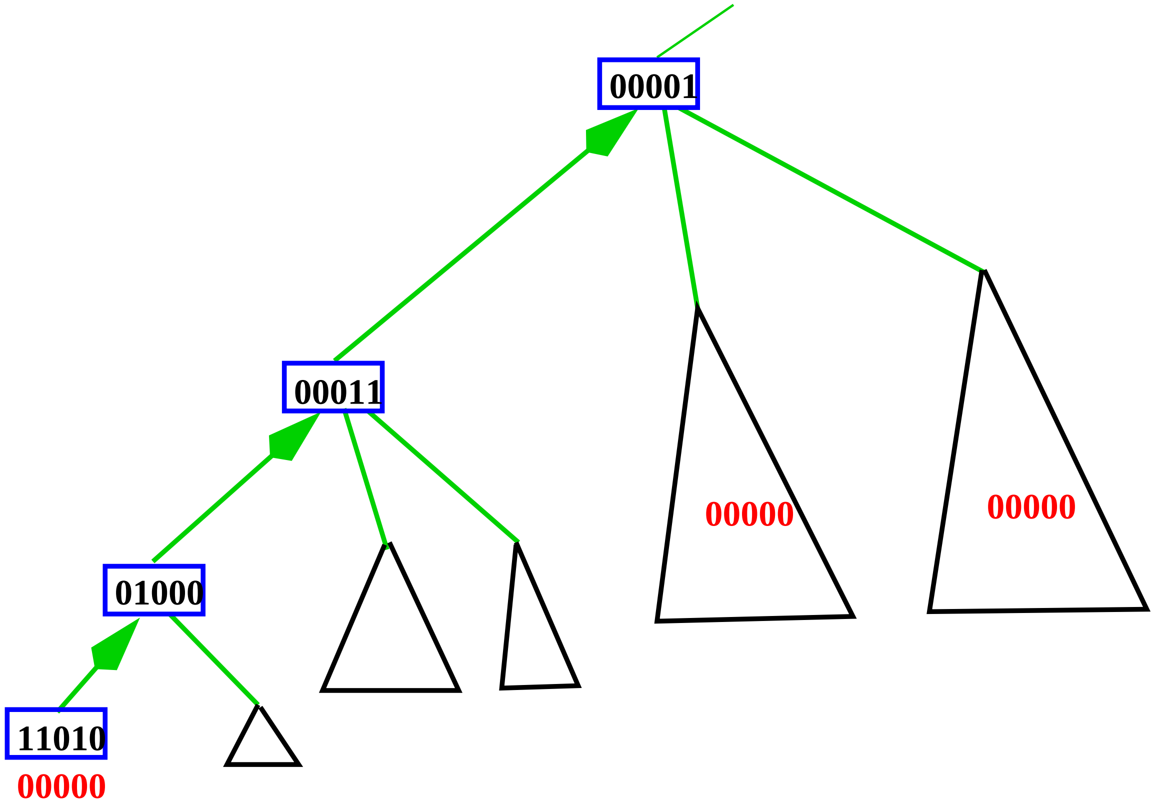
# Pointer Lists




Tree for object 000

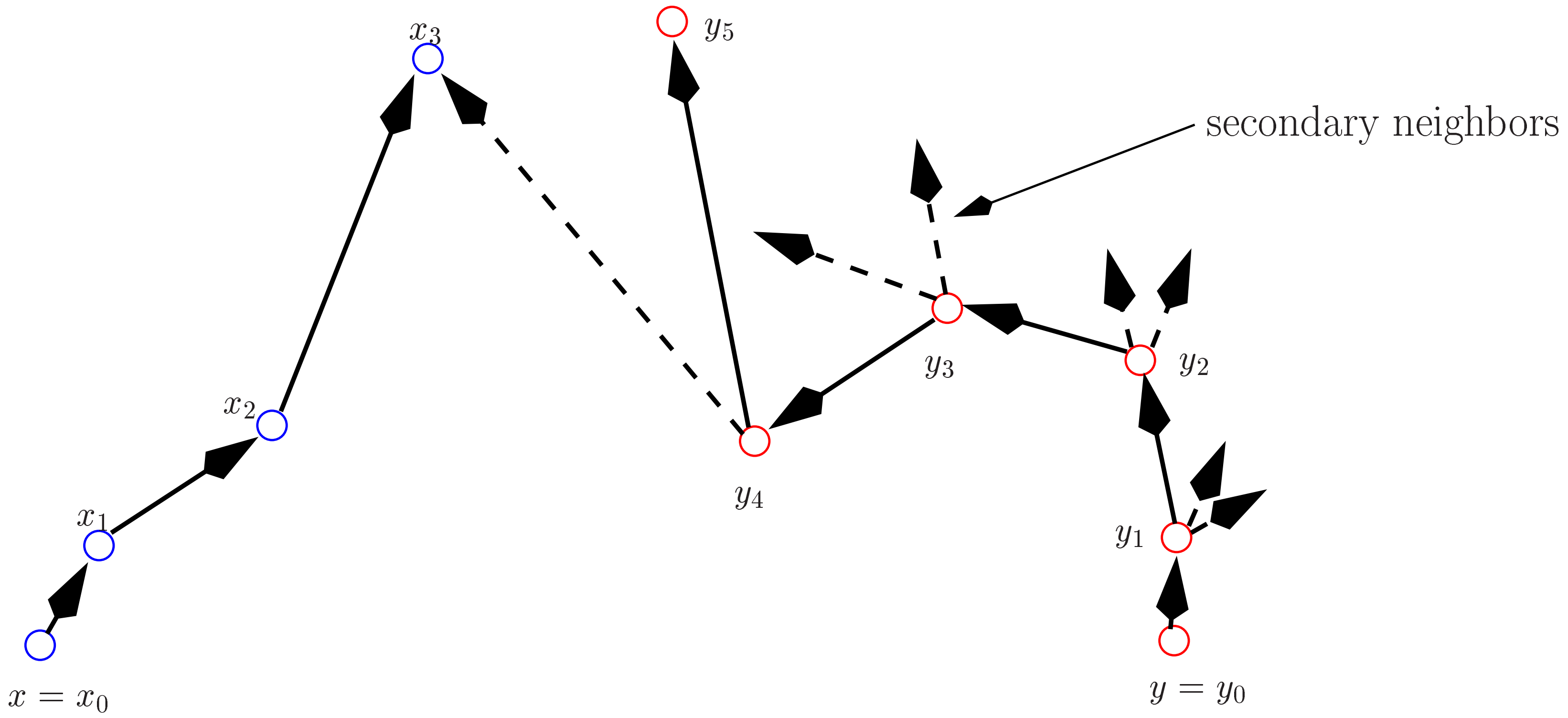
- For each object, the list contains a pointer to a copy of the object (if one exists) in the subtree rooted at the node.

# Inserting an Object Copy




- Follow the search path along the tree, updating pointer lists, until a pointer to the object found.

# Accessing an Object

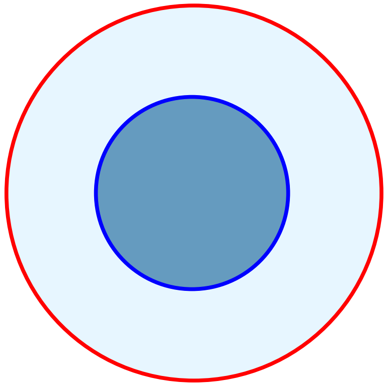



- Object inserted at $x$ and then requested at $y$.
- Follow the search path, querying both primary and secondary neighbors until a pointer to object found.

# Properties of the Tracking Scheme

- Scalable: The overhead incurred due to control information is small.
  - The neighbor table is small; by construction, the total number of "neighbors" of a node is $\log n$.
  - Due to the randomized ID assignment, the total set of pointers is evenly distributed.
- Efficient under certain assumptions about the communication cost function.
  - The expected access cost is within a constant factor of the optimal cost.
  - The expected number of nodes that need to be updated on a join/leave is $O(\log n)$.

# Restricted Class of Cost Functions



- For every node $x$ and real $r \geq 1$, the ratio of # nodes within cost $2r$ of $x$ to # nodes within cost $r$ is bounded from above and below by constants.

$$\min\{\delta N(x, r), n\} \leq N(x, 2r) \leq \Delta N(x, r).$$

- Applies to fixed-dimension meshes, constant-degree trees, and fat-trees.
- Purely "local" restriction and does not require any hierarchical decomposition of the network or regular topology.

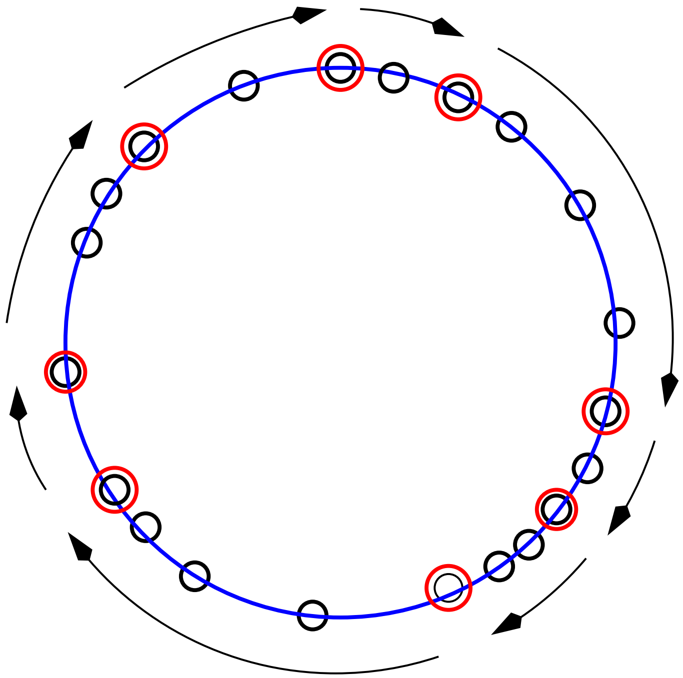
# Limitations

- The efficiency claims hold for a restricted class of cost functions.
- Does not consider dynamic load on the nodes.
- The overhead of forwarding the requests through several nodes may be significant.
- Join/leave:
  - No distributed scheme for handling these operations.
  - In practice, the number of nodes affected by a join/leave may be large.

# Consistent Hashing and Chord

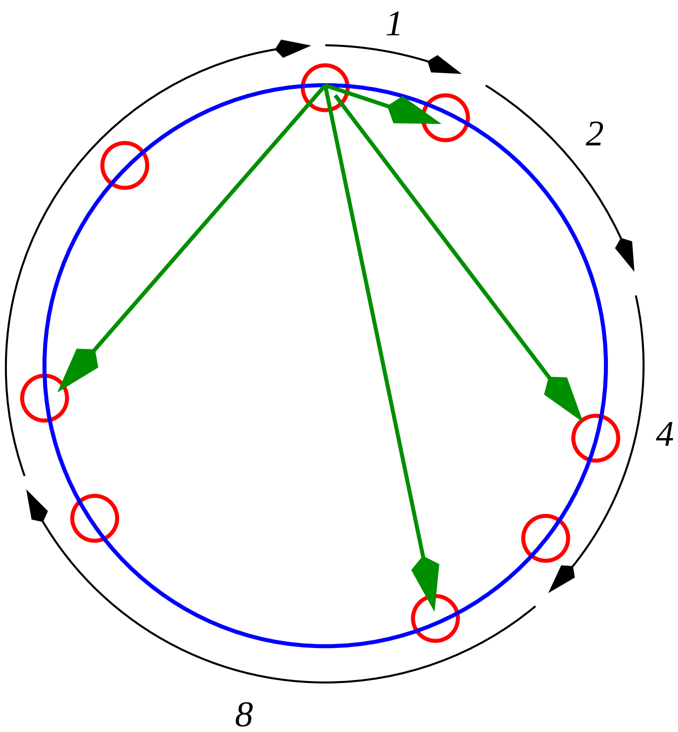
- A peer-to-peer lookup service [Stoica et al 01].
  - Using consistent hashing, map keys to nodes.
  - Each node has a small number of "neighbors" for forwarding requests it cannot resolve.
- Adaptive to node joins/leaves.
- Correctness in presence of inconsistent forwarding information.

# Mapping keys to nodes



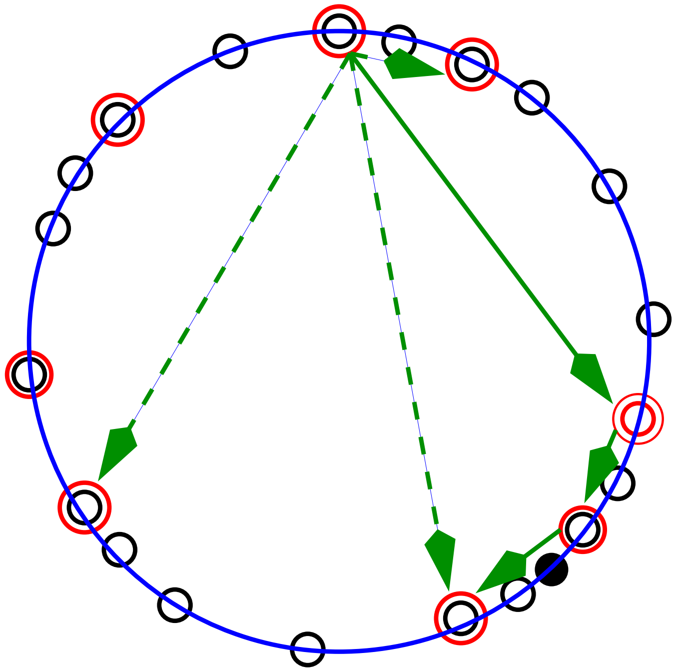
- If key and node IDs are selected uniformly at randomly, then asymptotically balanced load with high probability.
- One possible forwarding mechanism: if key information not stored, forward request to successor.

# Neighbors




- Number of neighbors for each node is at most $m$, the number of bits in the key identifiers.

# Looking up a Key



- Forward request for key to closest predecessor in the neighbor table.
- Number of hops is $O(\log n)$ whp.

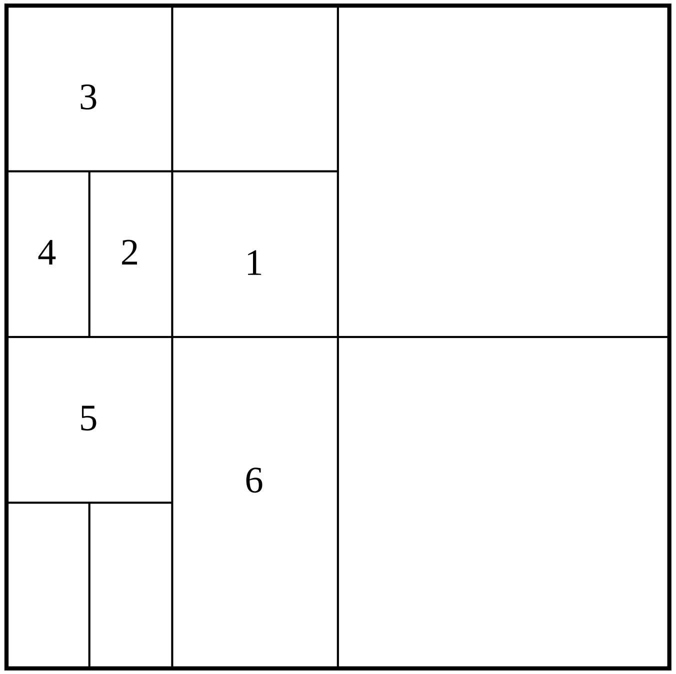
# Node Joins/Leaves

- New node $u$ has an existing node use the lookup procedure to find all $u$'s neighbors.
  - Number of communication steps is $O(m \log n)$ whp.
  - Can reduce to $O(\log^2 n)$ whp since if $m \gg \log n$, many of the intervals would be empty.
- Similarly can identify nodes whose neighbor tables need to include $u$ now, in $O(\log^2 n)$ communication steps.

# Content Addressable Network (CAN)

- A variant of the consistent hashing idea [Ratnasamy et al 01].
- A logical $d$-dimensional torus is the underlying space into which keys are mapped.
- The allocation of keys to nodes is given by the partitioning determined by the nodes.
- Each new node selects a random point and splits the zone which contains this point.
- When a node leaves, two adjacent zones are merged.

# Illustration of CAN

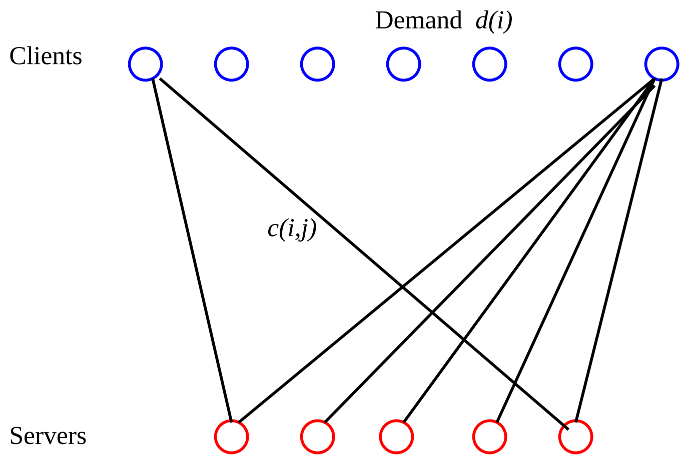



- Suppose new node 7's random choice is a point in zone 2.
  - Zone 2 is identified using the routing scheme already in place, starting from any node.
  - Zone 2 is split into two halves and 7's neighbors are {1, 2, 4, 5}.

# Balancing Dynamic Load on Replicas

- The lookup algorithms discussed thus far do not take into account current load while mapping requests.
- In P2P file sharing:
  - New copies of popular objects will automatically get created, hence keeping average load small.
  - However, the nodes holding the key-location associations may get overloaded.
- Also needed if new copies of objects are not being created or flash crowds arise in a "localized region".
- Questions:
  - How do we maintain dynamic load information in a distributed setup?
  - If we have all the load information, how do we assign the requests?

# An Online Assignment Problem




- Assigning unit demand at client $i$ to server $j$ incurs a cost of $c(i, j)$ and increases load on $j$.
- Two possible load models:
  - There is a capacity $C_j$ for server $j$.
  - There is a function $f_j$ of load that gives additional cost for each unit demand that is served by $j$.

# Related Variant

- If we assume that the $f_j$'s are concave, then we have a variant of the assignment problem discussed earlier.
  - Each demand is assigned to a single server.
  - The problem is NP-hard, by a reduction from set cover.
- Generalization of facility location.