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

H.V. Jagadish [1,2], Beng Chin Ooi [2,3], Quang Hieu Vu [3]

[1] Department of Electrical Engineering and Computer Science,
University of Michigan, MI USA
[2] Department of Computer Science,
National University of Singapore, Singapore
[3] Singapore-MIT Alliance, 4 Engineering Drive 3,
National University of Singapore, Singapore
jag@eecs.umich.edu, ooibc@comp.nus.edu.sg, hieuvq@nus.edu.sg

Abstract

We propose a balanced tree structure overlay on a peer-to-peer network capable of supporting both exact queries and range queries efficiently. In spite of the tree structure causing distinctions to be made between nodes at different levels in the tree, we show that the load at each node is approximately equal. In spite of the tree structure providing precisely one path between any pair of nodes, we show that sideways routing tables maintained at each node provide sufficient fault tolerance to permit efficient repair. Specifically, in a network with N nodes, we guarantee that both exact queries and range queries can be answered in $O(logN)$ steps and also that update operations (to both data and network) have an amortized cost of $O(logN)$. An experimental assessment validates the practicality of our proposal.

1 Introduction

Peer-to-Peer (P2P) systems have become popular recently. The central strength of P2P systems is the capability of sharing resources so that larger costly servers can be replaced by systems of smaller computers. The biggest challenge in building an effective P2P system is in tying together these multiple autonomous computers into a cohesive system. This is usually done by means of a logical "overlay network" used to organize the data managed by these computers, which represent nodes in this overlay network. Various topologies have been suggested for this network, including a ring [7], and a multi-dimensional grid [13]. With several of these overlays, it is well-known how to build "distributed hash tables" across nodes in a P2P system.

In the database world, B-trees occupy a central place, and the value of tree structures in general is very well appreciated. Yet, no overlay network proposed so far has a tree topology – and with good reason: in a typical (centralized) tree, nodes near the root are much more frequently accessed than nodes near the leave: this sort of skew in access load is typically not acceptable in a peer-to-peer system. In this paper, we propose a tree-structured overlay network for a peer-to-peer system that does not have a substantial skew in access load.

The overlay network we propose is based on a binary balanced tree structure in which each node of the tree is maintained by a peer. Each peer in the network stores a link to its parent, a link to its left child, a link to its right child, a link to its left adjacent node, a link to its right adjacent node, a left routing table to selected nodes on its left hand side at the same level, and a right routing table to selected nodes on its right hand side at the same level. We call our proposed structure BATON, for BAlanced Tree Overlay Network. While the tree structure is binary, it has scalability and robustness similar to that of the B-tree. An immediate benefit of a tree structured overlay network is convenient support for range queries, which cannot be supported by conventional distributed hash

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

tables.

There have been notable recent works to address range maintenance and querying [3, 4, 5, 15], but these schemes are based on standard overlay networks, and hence are quite complex. In the P-tree [3], each P2P node represents a leaf node in the B$^+$-tree, and each node maintains the path from the index root to the leaf node. These P2P nodes are distributed on the Chord overlay network. Explicit maintenance of the paths is expensive and is prone to inaccuracy introduced by changes on the B$^+$-tree structure. In [4], single dimensional range partitioning is examined independent of an underlying P2P network. However, most networks, such as CAN[13] and Chord[7], are based on uniform hashing upon which their good performance is achieved. Consequently, they are not effective for supporting data partitioning and retrieval based on ranges.

Our paper makes following contributions.

- To our knowledge, we are the first to build a P2P overlay network based on a balanced tree structure. In consequence, both exact match and range queries are efficiently supported.

- When a node joins or leaves the system, like other P2P systems, our system takes log N steps for finding a place for the joining node or for finding a node to replace the leaving node. However, it takes only $O(logN)$ cost for updating the routing table, which is more efficient than other P2P systems, which usually require $O(log^2N)$ for updating routing tables. This difference in asymptotic cost can become significant for networks with a large number of nodes, N.

- Load balancing and Range Partitioning: Our system does not require knowledge of stored data range in advance: it can adjust this range dynamically at each node. The same mechanism also permits load balancing, with overloaded nodes transferring part of their contents to other nodes. We show that the asymptotic cost of such load balancing is low.

- Fault Tolerance: A tree, by definition, has exactly one path between any pair of nodes. The small number of additional links stored in our network suffice to provide efficient recovery in the event of a node or a link malfunctioning. Specifically, we show that the network remains connected even with a large number of failures.

The rest of the paper is organized as follows: In Section 2 we present related work. In Section 3 and 4, we introduce our system architecture and system operators in detail. In Section 5 we present the performance study. Finally, we conclude in Section 6.

2   Related Work

Data partitioning and searching over multiple sites are well researched in the context of distributed databases [12, 9]. However, partitioning and searching strategies cannot be applied to fully distributed P2P networks where there is no global index and no guarantee on the uptime of individual system. In what follows, we will review recent related work in P2P systems.

CHORD [7], CAN [13], Pastry [14], and Tapestry [16] are four of the best-known P2P systems. Each of these implements a distributed hash table, which is efficient for exact queries but is not well suited for range queries since hashing destroys the ordering of data. To rectify this, Gupta et al [5] proposes a P2P system based on Locality Sensitive Hashing in which similar ranges are hashed to the same peer with high probability. However, these methods can only help to get approximate answers. Another way is including the ranges into hash functions proposed by Sahin et al [15] so that the system can return a superset of the range query. Nevertheless, exact search is highly inefficient. SkipList based systems such as SkipNet [6] and SkipGraph [2] can support range queries but they do not guarantee data locality and load balancing in the whole system.

The closest works to ours are P-Tree [3], P-Grid [1], and [11]. The P-Tree structure is based on B$^+$-tree structure and uses CHORD as its overlay routing architecture. Each node in the system stores the left-most root-to-leaf path of its corresponding B$^+$-tree. Data is only stored in leaf nodes, and these leaf nodes form a CHORD ring. P-tree guarantees $logN$ search for both exact query and range query. When a node joins the network, in addition to the $logN$ cost of searching its predecessors in CHORD ring and the log^2N cost of updating the routing tables, there is a large cost of getting tree structures from its predecessors to build the tree branch for it. Moreover, to check data consistency of new join nodes, it requires a special process run periodically in other nodes in the system. Like other systems based on CHORD, its performance degrades when the data is skewed. P-Grid [1] is a binary prefix tree structure in which each node in the tree maintains references to other nodes, that have the same prefix of length l, but a different value at position $l+1$, for the key they are responsible for. The multiway tree [11] is also a tree structured overlay in which each node in the tree is maintained by a peer and has a link to its parent, its children, its siblings, and its neighbors. [11]'s tree structure is neither B$^+$-tree nor binary tree; it is a multiway tree structure with no constraints on the fan-out and hence a node can have as many children as possible. Searching entails hopping from the query node to the node containing the answer by following the links, one by one. As each node in the multiway tree is connected only to the parent, and left and right sibling or neighbor sub-
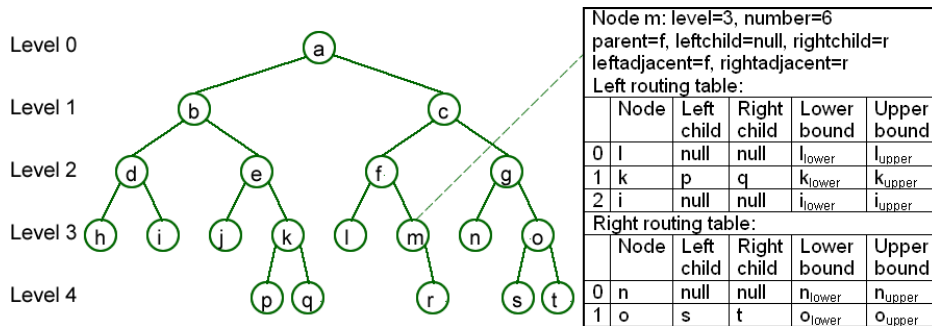
Figure 1: Binary balanced tree index architecture

trees, it is prone to network failure. In both P-Grid [1] and [11], the tree is not balanced if the data is skewed and in the worst case, the tree structure can become a linear linked list in which there is only one node at each level. Hence, the search process cannot be guaranteed within $logN$ steps. Compared to existing tree-based network structures, our proposed structure, BATON, is self adjusting to data skew and is 'height-balanced', and maintains vertical and horizontal routing information for efficient search and fault tolerance.

3 BATON Structure

The overlay network in BATON is a binary balanced tree structure as shown in Figure 1.

Definition 1: *A tree is* balanced *if and only if at any node in the tree, the height of its two subtrees differ by at most one.*
It has been shown that a binary balanced tree with N nodes has height no greater than $1.44logN$ [8].

We associate with each node in the tree a *level* and a *number*. The level of the root is 0, its immediate children are at level 1, and so on. The level of any node is one greater than the level of its parent. Hence, the maximum level number in the tree is one less than height of the tree. At level L there are at most 2^L nodes in a binary tree. We number these 2^L positions from left to right, from 1 until 2^L, within each level, whether or not there is a node currently instantiated at that position. Thus, the level and number together precisely determine the location of a node in the binary tree. It is straightforward to use these to determine structural relationships, if any, between a given pair of nodes – not just parent-child and ancestor-descendant relationships, but also siblings, neighbors, and so forth.

We will find it useful to have a linear ordering of the nodes in the tree, and for this purpose we use an *in-order* traversal. Given a node x, we say that the node immediately prior to it in the traversal is *left adjacent* to it, and the node immediately after x is *right adjacent* to it. Note that adjacent nodes may be at very different levels. In fact, in a complete tree, every alternate node in the traversal is a leaf node, and every other alternate node is an interior node. Even

when the tree is not complete, it is easy to show that each interior node must have at least one adjacent node that is either a leaf node or an interior node with less than two children.

Each node in the tree typically maps to exactly one peer compute node in the peer-to-peer system. (There will be times when this mapping may not be one to one, at least temporarily; but, for the present, in terms of developing an intuition for the network, it is reasonable to state that each node in the conceptual binary tree corresponds to exactly one unique node in the peer-to-peer system). Each physical compute node has an IP address or other network ID associated, which can be used to locate the node and communicate with it. Thus we will think of each node having a logical id in terms of its level and number, and a physical id in terms of its IP address.

Each node in the tree maintains "links" to its parent, children, adjacent nodes, and selected neighbor nodes which are nodes at the same level. Maintaining links to parent, children and adjacent nodes simply means maintaining the physical id of the parent, of the left child, of the right child, of the left adjacent node, of the right adjacent node, if any. Links to selected neighbors are maintained by means of two special sideways routing tables: a *left routing table* and a *right routing table*. Each of these routing tables contains links to nodes at the same level with numbers that are less (respectively greater) than the number of the source node by a power of 2. The jth element in the left (right) routing table at node numbered N contains a link to the node at number $N - 2^{j-1}$ (respectively $N + 2^{j-1}$) at the same level in the tree. If there is no such node, an entry is still made in the routing table, but marked as null. A routing table is considered *full* if all valid links are not null. For example, consider node h in Figure 1. Its left routing table has no valid links, and its right routing table contains neighbor links to node i, j, and l which are 2^i nodes away from h ($i = 0, 1, 2$). This structure has some similarity with Chord, except that it is on a straight line rather than on a circle, and routing table entry carries additional information beyond just the target

IP address, and some links could be null.

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.*

Proof: Consider the addition of a node to a tree that is balanced. Let this new node be added as a child of node x. Let node x be at level L, and the new node at level $L + 1$. The resulting tree could become imbalanced if at any ancestor of node x, the depth of the left and right subtrees differs by more than 1 as a result of this new node addition. Consider the ancestor y of x at level i. Without loss of generality suppose that x is in the left subtree of y. The depth of this left subtree may have changed from L to $L + 1$ as a result of the node addition. (If the depth of the left subtree was already $L + 1$ or greater, then it does not change as a result of the node addition, and no imbalance can result). But since the right routing table of x is full, there must be an ith entry in this table to a node in the right subtree of y, and furthermore this node is at level L. Therefore the right subtree of y has depth at least L, and a change of depth of the left subtree to $L + 1$ does not violate balance. Applying the same argument to every ancestor of X in the tree, we can establish that the tree remains balanced after any node addition.

Now consider deletion from the tree of a node u that is a child of node x at level L. This deletion may cause an imbalance in the tree rooted at any ancestor y of x if the depth of the x subtree changes from $L + 1$ to L while the other subtree has depth $L + 2$. Without loss of generality, suppose that x is in the left subtree of y. There must exist a node z in the right routing table of x that is in the right subtree of y. Node z is also at level L. Suppose the depth of the right subtree of y is $L + 1$ or less, we are done – no imbalance is created. Suppose the depth is $L + 2$ or greater. Consider two cases.

Case 1: There is a node v, child of z, that is in the right routing table of u and has a child. By the requirement of the theorem, the deletion of u is not permitted if z has any children. So no imbalance can be caused.

Case 2: There is no such node v. This means that the any node at level $L+2$ in the right subtree of y has a parent at level $L + 1$ which has an entry in its left neighbor routing table of a node w in the left subtree of y that is different from u. Node w is at level $L + 1$, so the departure of node u does not change the depth of the left subtree of y. Hence again, no imbalance is caused.

Making the above argument for each ancestor y of x, we show that tree balance cannot be destroyed by node deletion that is subject to the theorem condition. □

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

Proof: Let N_x be the number of node x and let the parent of x be w. Without loss of generality, let x be the right child of its parent w. Then N_x is even. We must have $N_w = N_x/2$. Similarly, let the parent of y be z. We must have $N_z = N_y/2$ if N_y is even, and $N_z = (N_y + 1)/2$ if N_y is odd.

Case 1: Suppose y is at least distance 2 from x. Then $N_y = N_x \pm 2^k$ for some integer $k > 1$. N_y is thus guaranteed to be even, if N_x is even. It follows that $N_z = N_w \pm 2^{k-1}$, meaning that z has a link from w.

Case 2: Suppose y is at distance 1 from x, and is its sibling. Then x and y have the same parent, and we are done.

Case 3: Suppose y is at distance 1 from x, but is not a sibling. Since x is the right child of its parent, y must be the right neighbor of x. That is $N_y = N_x + 1$, and is odd. We then compute $N_z = (N_y+1)/2 = (N_x + 2)/2 = N_w + 1$. Since z has a number one greater than w, the latter must link to it.

It is easy to see that there are at most L entries in a left (right) routing table at level L. Therefore the total number of entries is $O(logN)$ - the same asymptotic bound as for Chord, though in the worst case number of entries could be twice as many as in BATON, and each entry also is larger. □

3.1 Node Join

A node wanting to join the network must know at least one node inside the network and sends a JOIN request to that node. There are two phases in a new node joining the network. The first is to determine where the new node should join. The second is actually including it in the network at a specified place. We consider each in turn.

When a node receives a JOIN request, if it has both its left routing table and its right routing table full while it has less than two children, it can accept the new node as its child. Otherwise, it needs to forward the JOIN request to other nodes as in the join algorithm described below.

For example, assume that node u wants to join the network and it sends a JOIN request to node b as in Figure 2. b then forwards the request to p, which is its adjacent node. As p's routing tables are not full, it forwards the request to its parent j. In turn, j checks its routing tables and forwards the request to the neighbor node n, which doesn't have enough children. Finally, n accepts u as its child.

Analyzing the algorithm, suppose that an adjacent link is traversed to a leaf node w. Either w is able to accept the new node as a child, or has an incomplete neighbor table. In the latter case w forwards the request to its parent, which can locate in its neighbor table a node v that is the parent of a missing neigh-

Algorithm: *join(node n)*
If (Full(LeftRoutingTable(n)) and
 Full(RightRoutingTable(n)) and
 ((LeftChild(n)==null) or (RightChild(n)==null))
 Accept new node as child of n
Else
 If ((Not Full(LeftRoutingTable(n))) or
  (Not Full(RightRoutingTable(n))))
  Forward the JOIN request to parent(n)
 Else
  m=SomeNodesNotHavingEnoughChildrenIn
   (LeftRoutingTable(n), RightRoutingTable(n))
  If (there exists such an m)
   Forward the JOIN request to m
  Else
   Forward the JOIN request to one of its
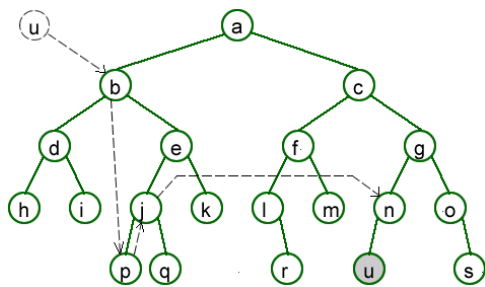   adjacent nodes
  End If
 End If
End If

Figure 2: A new node joins the network

bor of w. Node v can now accept the new node as a child, unless its own neighbor table is not full, in which case it forwards the request to its parent. Since the height of the tree is $O(logN)$, the request cannot be forwarded up in this manner more than $O(logN)$ times. All other directions of forwarding only add constant terms. Thus we have a bound of $O(logN)$ messages to locate a spot for a new node to join. Furthermore, the algorithm specifically seeks out leaf nodes, and parents of nodes with incomplete neighbor tables, which must all be leaf nodes due to Theorem 1. Specifically, ancestor nodes are never required, and there is no involvement of the root other than just as an ordinary node. As such, we expect that the load is not disproportionately applied to the root.

When a node x accepts the new node y as its child, it splits half of its content to its child. In other words, the range associated with x is partitioned between itself and its new child. In addition, if y is accepted as x's left child, x also sends its left adjacent link, which points to z, to y, and updates its left adjacent link with y. y then creates its left adjacent link pointing to z and its right adjacent link pointing to x, and

also notifies z that z should update its right adjacent node with y instead of x as before. Similarly, if y is accepted as x's right child, x's right adjacent link is transferred to y. Finally, node x contacts all neighbor nodes in its left and right (sideways) routing tables, asking them to inform their relevant children about y, and in turn responding with information regarding their relevant children that y will require. This whole process requires $O(logN)$ messages and $O(logN)$ responses. Specifically, x needs to send maximum $2L_1$ messages to its neighbor nodes, where L_1 is level of x. These neighbors nodes need to send maximum of $2L_2$ messages to their children which in turn need to send $2L_2$ messages to respond to the new node, where L_2 is level of the new node. The new node y needs to send only one message to one of its adjacent nodes. Therefore, the maximum number of messages required for updating routing tables is $2L_1+2L_2+2L_2+1 < 6logN$.

3.2 Node Departure

Only leaf nodes may voluntarily leave the network, and only if their departure will not upset the tree balance. In other cases, a node that wishes to leave the network must find a replacement for itself, which will be a leaf node whose absence does not affect the tree balance. We consider these cases in turn.

If a leaf node x wishes to leave the network, and there is no neighbor node in its routing tables with children, it can leave the network without affecting the tree balance because the requirement in Theorem 1 is still satisfied at its neighbor nodes. In this case, x has to transfer all its content, and its range of index values it is in charge of to its parent, its left adjacent link if it is a left child or its right adjacent link if it is a right child, and send LEAVE messages to its neighbor nodes to update their routing tables. The parent node of x after receiving the content from x also needs to send messages to neighbor nodes to notify them of its new content and children. It also notifies affected adjacent link node to update the corresponding adjacent link with itself. Thus, the total number of messages required in this case is $2L_1 + 2L_2 + 2 < 4logN$, where L_1 and L_2 are levels of x's parent node and x node. If a leaf node wishes to leave the network, and there are neighbor nodes in its routing tables with children, it needs to find a node to replace it by sending a FIND-REPLACEMENT request to a child node of one of its neighbor nodes. If a non-leaf node wishes to leave the network, it finds a node to replace it by sending a FINDREPLACEMENT request to one of its adjacent nodes, which is a leaf node, or as deep as possible. The find replacement algorithm is described as below

As the process of finding a replacement node always goes down, it takes at most as many steps as the height of the tree which is $O(logN)$. For example, consider node b in Figure 3. If it wants to leave the network, it has to find a leaf node as replacement. b creates a

Algorithm: *find replacement node (node n)*
If (LeftChild(n)!=null)
   Forward the request to LeftChild(n)
Else If (RightChild(n)!=null)
   Forward the request to RightChild(n)
Else
   m=SomeNodesHavingChildrenIn
     (LeftRoutingTable(n), RightRoutingTable(n))
   If (there exists such an m)
     Forward the request to a child of m
   Else
     Come to replace the leave node
   End If
End If

FINDREPLACEMENT request and sends to its adjacent node j. j checks its routing tables and realizes that there are some neighbor nodes with children, j therefore forwards the request to r, which is a child of a neighbor node of j. At r, as it does not have any child, and there is no neighbor node with children, r can replace b safely. As illustrated, BATON adapts itself to node departure and continues to maintain its height-balanced property.
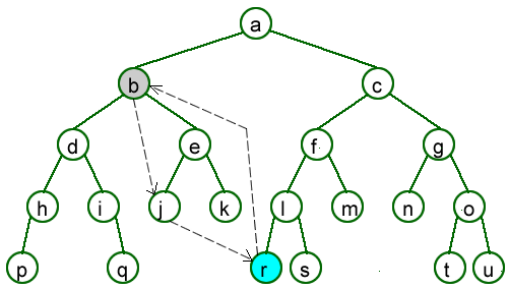

Figure 3: An existing node leaving the network

Before a node y replaces node x which is leaving the network, it needs to notify its neighbor nodes and its parent of its leaving as in the previous case, which takes $4logN$ steps. In addition, all nodes with links to x must be informed to change the physical (IP) address of the link to point to y instead of x. This can easily be done by using information received from x. Specifically, the original parent of node x (now y) needs to send $2L_1$ messages to its neighbor nodes to notify its new replacement child at level L_1. y needs to send $2L_2$ messages to its new neighbor nodes, where L_2 is its new level, and 2 messages to its children and 2 messages to its adjacent nodes. Thus, the maximum number of messages required to update routing tables to reflect changes is $8logN$.

3.3 Node Failure

Sometimes, a node may fail, or depart suddenly. In such a case, some nodes wishing to access the departed node x, will discover the address unreachable. These nodes must report this failure to node y, the parent of x, which now has the responsibility of managing the departure of x. Node y makes use of links maintained in its own routing tables and quickly regenerate the left and right routing tables of x by contacting children of nodes in its own routing tables. These children can also help to locate the children of x if any. Node y, the parent of node x can now initiate a "graceful departure" for its child node y that has left abruptly, following the protocol described in the preceding section. Since all of x's routing information has been regenerated at y, the algorithm described above works with minor modification.

3.4 Fault Tolerance

We have described above how the failure or abrupt departure of a node can be handled gracefully. The repair operation is the same as in node departure, requiring only $O(logN)$ messages, but it takes non-zero time. In this section we show how the network can continue to operate, routing around the missing node, in the mean time.

There are two axes along which messages are routed in BATON: the sideways axis, through the left and right routing tables, and the up-down axis, through parent, child and adjacent links. The former is naturally fault tolerant, since there is a Chord-like logarithmic expansion of links, and therefore a large number of alternative paths between any pair of nodes. The latter is rendered fault tolerant because a node can go to a neighbor of the parent, find a child of that node, and then connect back to the child node, thereby reconstituting a missing parent child link. Thus far we have considered failure of only a single node. If two (or more) nodes fail, we have two possibilities to consider. If the failed nodes have a parent-child relationship, we can still apply the same technique as described above – traveling via neighbor nodes. If the failed nodes do not have a parent-child relationship, then their failures can be corrected independently, and there is no additional complication because of the temporal simultaneity of their failure. In a special case, even if all nodes at the same level fail, the tree is not partitioned since adjacency links can be used to route across the gap. Contrast this with the brittleness of multiway tree[11].

3.5 Network Restructuring

In the description above, joining nodes are forced away to other parts of the tree while leaving nodes have to find replacement nodes if they cause the tree to become imbalanced. Sometimes, when the node joins(leaves) is part of a load balancing effort (see 4.5), this redirection may not be permitted. An alternative is to restructure the system to achieve balance. The restructuring is akin to a rotation in an AVL tree and is described here.
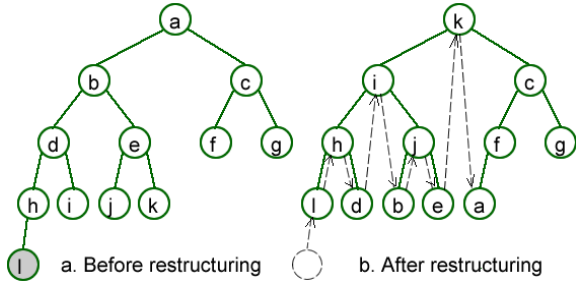
Figure 4: Network restructuring after a node joining

Figure 5: Network restructuring after a node departure

When a node x accepts a joining node y as its child and detects that Theorem 1 is violated, it initiates the restructuring process. Without loss of generality, suppose that this restructuring is towards the right. Assume that y joins as x's left child. To rebalance the system, x notifies y to replace its position, and notifies its right adjacent node z that x will replace z's position. (If y joins as x's right child, then x itself remains untouched, y directly replaces z). z then checks its right adjacent node t to see if its left child is empty. If it is, and adding a child to t does not affect the tree balance, z takes the position of t's left child as its new position and the restructuring process stops. If t's left child is full or t cannot accept x as its left child without violating the balance property, z occupies t's position while t needs to find a new position for itself by continuing to its right adjacent node. Consider the example in Figure 4. Suppose l joins the network as left child of h and the joining violates the tree balance property (as shown in Figure 4a). The restructuring process is initiated at h in which l replaces h; h replaces d; d replaces i; i replaces b; b replaces j; j replaces e; e replaces k; k replaces a, and finally a becomes f's left child because f can accept a child without causing the tree to become imbalanced. The tree now is balanced again, as illustrated in Figure 4b.

When a *leaf* node x leaves the network and causes the tree to be imbalanced, its parent y starts the restructuring process (non-leaf node still needs to find a replacement node). Without loss of generality, consider a left restructuring. Assume that x is y's right child. To rebalance the tree, y has to replace x, and its left adjacent node z has to replace y. (If x is the left child of y, then z can directly replace x, and y remain untouched.) If z's move does not upset the tree balance, the restructuring process stops. If z's move does violate the balance property, we use its left adjacent node t to replace its position, and recursively find the replacement node for t. For example, assume that g leaves and makes the system imbalanced as shown in Figure 5a. The restructuring process is started at c in which c replaces g, f replaces c, a replaces f, and finally k replaces a. The process stops at a because a's move does not cause any loss of balance. Figure 5b shows the balanced structure after restructuring.
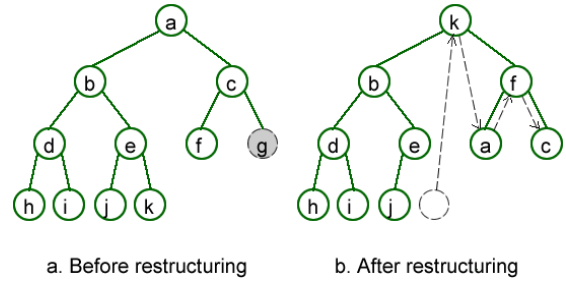
No data movement is required due to network restructuring. However, several nodes change their position in the tree, affecting their level and number, and affecting their routing tables. For each such node, adjusting the routing table requires $O(logN)$ effort. Thus, the more nodes that participate in the restructuring process, the more effort is required for updating routing tables.

4 Index Construction

In the previous section, we have described an overlay network structured as a binary balanced tree. In this section, we show how to use such an overlay network to build an effective distributed index structure, very similar in spirit to an AVL tree.

We assign to each node, both leaf and internal, a range of values. We record for each link the range of values managed by the node at the target of the link. Whenever this range changes, the link has to be modified to record the change.

The range of values directly managed by a node is required to be to the right of the range managed by its left subtree and less than the range managed by its right subtree. In other words, unlike B^+-trees, internal nodes in the tree themselves also manage a range of data values directly.

With this, it is easy to see how the BATON overlay structure immediately behaves like an index tree. Indeed, the index is structurally similar to the main-memory index called T-Tree[10], which was designed to reduce number of pointers and in-memory pointer chasing.

4.1 Exact Match Query

For an exact match query issued or received at node x, the node will first check its own range. If it is within the current range, the local index is searched for the value, and the search stops. Otherwise, x routes the query to the destination node as described below in the search exact algorithm.

Algorithm: *search exact(node n, query q, value v)*
If ((LowerBound(n)<=v) and (v<=UpperBound(n)))
  q is executed at x [1]
Else
  If (UpperBound(n)<v)
    m=TheFarthestNodeSatisfyingCondition
      (LowerBound(m)<=v)
    If (there exists such an m)
      Forward q to m
    Else
      If (RightChild(n)!=null)
        Forward q to RightChild(n)
      Else
        Forward q to RightAdjacentNode(n)
      End If
    End If
  Else
    //A similar process is followed towards the left
  End If
End If

We now illustrate the search using Figure 6. Suppose node h wants to search for data that is stored in node c. Since the searched for value is greater than h's upper bound, it checks its right routing table and forwards the search request to node l, which is the rightmost node having the lower bound less than the searched value. l then checks its right routing table and forwards the request to m. At node m, as it cannot find any neighbor node to forward the request, it forwards the request to its right child r. Finally, r forwards the request to c, which is the destination node.
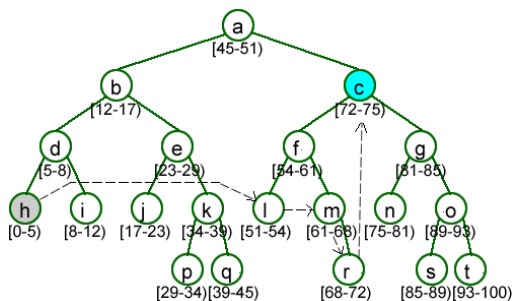

Figure 6: Exact match query search

When a node x wants to search for an exact value, if x is the root, the search request is always forwarded downward to the destination node whose range of index values contains the searched value. Thus, the maximum number of steps of processing is the height of the tree – $logN$. If x is not the root, without loss of gener-

---

[1]If there is a large number of duplicates in a partition search key value, the corresponding index entries may be distributed across multiple tree nodes. In such a case, x is one of these nodes found by the exact search algorithm. Adjacent node links must be used to navigate to the other index nodes.

ality, assume that the request node is on the left side of the tree. We consider two cases of the destination node. In the first case, if the destination node is the root, following the algorithm, the search request is always forwarded to the right most node r of the left subtree, and from there it is forwarded to the root, which is the right adjacent node of r. The cost of forwarding the request to r is $logN - 1$, which is the height of the left subtree, because for each forwarding, be it to the neighbor node, right child, or right adjacent node, the search space is always reduced by half. Thus, the maximum number of steps is also $logN$. In the second case, if the destination node is on the right side of the tree, during the search process, it takes one step to forward the search request from a node in the left subtree to a node in the right subtree via its routing table. Depending on the searched value, this step can happen early or later in the search process. However, if it happens later, previous search steps still help to reduce the search space of the right subtree by half. Thus, the total steps is also $1 + (logN - 1) = logN$, where $logN - 1$ is the cost of searching in the right subtree. Our algorithm shows that the search request is always forwarded via neighbor nodes or child nodes. The request is only needed to forward to higher level nodes in two cases: the higher level node contains the searched value, or the processing node does not have two children (a leaf node or a node near the leave). *This property clearly helps the root to avoid receiving more requests than other nodes.*

4.2 Range Query

A range query proceeds exactly in the same manner as a point query, with only one difference: instead of looking for the data range at a node including the searched value, we now look for an intersection with the searched range. Once an intersection is found, we have at least partial answers for the range query. We then proceed left and/or right to cover the remainder of the searched range.

As in the case of a point query, it takes $O(logN)$ steps to find the first intersection. Thereafter it is a cost of $O(1)$ for each additional node to be visited. Therefore, to answer a range query, with the range covering X nodes, we require $O(logN + X)$ steps.

4.3 Data Insertion

When data is to be inserted, we first follow the search process for exact match query to find the node where this data should be inserted, and then perform the insertion. However, for the left most and right most nodes, their range may need to be adjusted if the inserted data value is outside the current range. If the left most node receives an INSERT request and the inserted value is still less than its range of values, it expands its range of values to the left so that it can cover the newly inserted value (a node knows that it's

the left most node if its number is one and it does not have left child). Similarly, if the right most node receives an INSERT request and the inserted value is greater than its range of values, it expands its range of values to the right, and accept the new inserted value. In these special cases, it takes additional $logN$ step for updating its routing tables. The cost of locating node to insert new data is $O(logN)$ as in the exact match query search process.

4.4 Data Deletion

To delete existing data, we locate the node that manages this data value, and delete the data. The cost is exactly that of the search, namely $O(logN)$.

4.5 Load balancing

We would like to distribute the computational load evenly across all nodes in the peer system. This load can be estimated in terms of number of queries or number of messages. Typically, the larger the range covered by a node, the more the number of data items managed by a node, and hence the more its load. The load balancing process allows the node to split part of its range to other nodes or acquire additional range from other nodes. The goal is to adjust the data range to (roughly) equalize the workload. Note that this doesn't mean data ranges are all equal.

Load balancing based on simple data migration between two adjacent nodes may not be sufficient to deal effectively with a very skewed dataset. Further, data migration may ripple through the network [9], and incur high total overhead. Thus, instead of doing load balancing with just adjacent nodes, we propose that a node only does load balancing with its adjacent nodes if it is a non-leaf node. If it is a leaf node, it can either load balance with its adjacent nodes or find another leaf node, which is lightly loaded node, to share its load. Specifically, when a leaf node becomes overloaded, it first tries to do load balancing with its adjacent nodes. If its adjacent nodes are also heavily loaded, then it finds[2] a lightly loaded node to do load balancing. Without loss of generality, let this lightly loaded node be to the right of the overloaded node. The lightly loaded node can pass its load to its right adjacent node. It then leaves the current position in the network and and re-joins as a child of the overloaded node, with forced restructuring of the network (to the left for the node leaving and to the right for the node joining) if necessary.

For example, assume that node g in Figure 7a. is overloaded and it identifies node f being a lightly loaded node. Node f passes the range to node c, and rejoins as a child of node g. The movement of node

---

[2]We could use a skip list structure for this as suggested in [4]. Our practical experience suggests that the neighbor tables suffice to locate a lighter loaded ode, even if not the lightest loaded node.

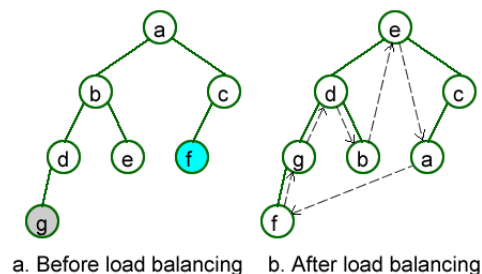a. Before load balancing   b. After load balancing

Figure 7: Load balancing with major restructuring

causes the overlay structure to become imbalanced and hence restructuring is invoked. In the load balancing process, nodes f replaces g, g in turn replaces d, d replaces b, b replaces e, e replaces a, and a takes over the original f position. The movement of nodes is illustrated with dashed line in Figure 7b.

Observe that the forced restructuring, in the worst case, involves a complete shift from the overloaded node position to the lightly loaded node position. More commonly, much smaller shifts are required, affecting only a few nodes at each end until suitable spots are found to accommodate the node departure and arrival respectively. In fact, the probability of the shift involving k nodes is exponentially decreasing with the value of k. With a little bit of analysis one can show that the amortized cost of load balancing per insertion or deletion is just $O(logN)$.

5 Experimental Study

We built a peer-to-peer simulator to evaluate the performance of our proposed system over large-scale networks. The simulator simulates P2P network by reading a schedule, which specifies actions inside the network. This schedule is generated randomly by a scheduler, which requires input parameters including number of nodes, number of data, and number of search queries. We use number of passing messages to measure the performance of the system.
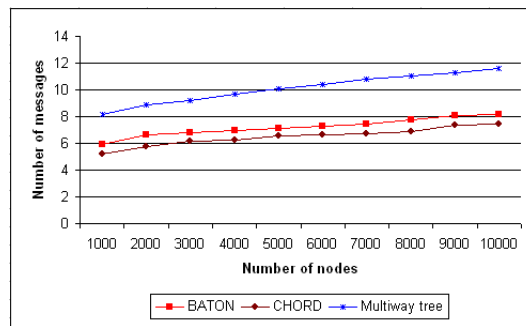
To evaluate the cost of operations, we test the network with different number of nodes N from 1000 to 10000. For a network of size N, 1000 x N data values in the domain of [1, 1000000000) are inserted in batches. For each test, 1000 exact queries, and 1000 range queries are executed, and the average cost is taken. To simulate different sequences of events (order in which nodes join and leave), the experiments are executed 10 times using 10 different sequences and the average is taken. For comparison purposes, we obtained CHORD [7] from its web site and implemented the multiway tree structure proposed in [11].

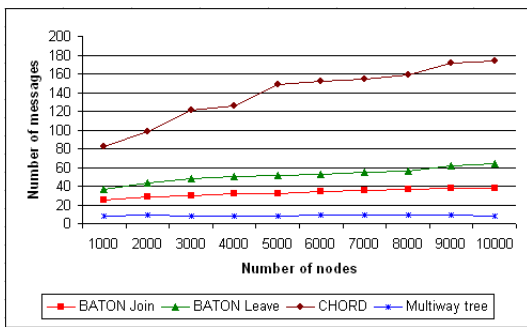### 5.1 Cost of Join and Leave Operations

Figure 8(a) shows average messages to find a destination node of the join operation and average messages to
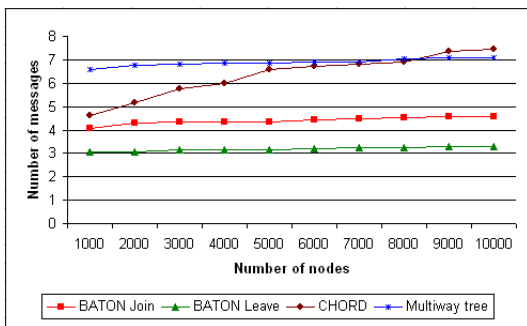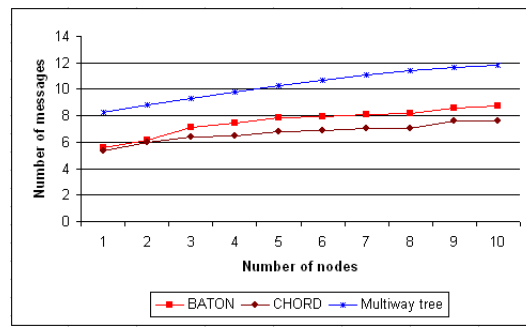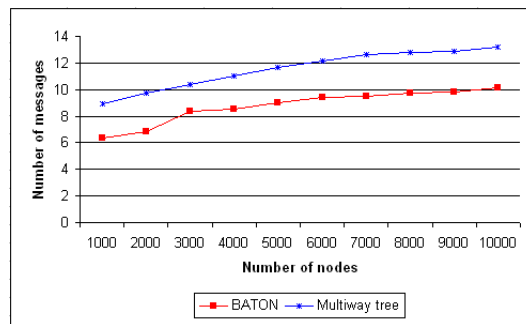
(a) Finding join node and replacement node

(b) Updating routing table

Figure 8: Average messages of node joining and leaving operations

(a) Insert and Delete operation

(b) Exact match query

(c) Range query

Figure 9: Average messages of insert, delete, and search operations

find a replacement node of the leave operation. The result is interesting. Even though the number of nodes in the network increases, the average number of messages of join and leave operations do not increase much. This is because no matter at what level the node is located, it takes only one step to forward the JOIN or LEAVE request to a leaf node. After that, the request is either forwarded upward to lower leaf nodes in case of the join operation or forwarded downward to higher leaf nodes in case of the leave operation. This is an important feature of BATON as the cost of these operations usually is equal to the distance from a lower node to higher node, which does not change much when the system grows. Moreover, as this distance is much lower than height of the tree, the cost of join and leave operations much lower than $O(logN)$. The figure also shows that the average number of messages of the leave operation is lower than those of the join operation because while the process of finding nodes to replace only needs to go down, the process of finding nodes to join sometimes needs to go horizontally in addition to going up. CHORD requires more messages, in comparison, and the number of messages increases linearly with network size. In the multiway tree system, if a node can have many children, the cost of join operation is low but the cost of leave operation is high because a departing node needs to get information from all of its children to select a replacement node; if a node has only a few children, the cost of join operation is in-

creased as there are higher chances for a join request to be forwarded to descendant nodes. In either case, the total number of messages required is large.

Figure 8(b) shows average number of messages required to update routing tables of join and leave operations. The experiment confirms our claim that our system significantly reduces the cost of updating routing tables compared with other systems such as CHORD, which require $log^2 N$ for updating routing tables. Compared with multiway tree system, our system takes higher cost because in the multiway tree system, a node only has links to its parent, its siblings, its neighbors and its children. Thus, the cost of updating routing tables depends on the number of children a node has. However, without sufficient routing tables, the multiway tree system must pay a high

price in search operations. Moreover, the system becomes vulnerable to link failure. In our case, in order to reduce cost of updating routing tables without increasing cost of other operations, we must keep more information in routing tables than other systems.

5.2 Cost of Insert, Delete, and Search Operations

Figure 9(a) shows the average number of messages needed for insert and delete operations while Figure 9(b) and Figure 9(c) respectively show the average number of messages required for exact match queries and range queries. The result shows that our system can support both insert and delete operations as well as exact match queries and range queries efficiently. The cost of insert, delete and exact match query operations in our system as a balanced tree is much lower than those of [11] as a multiway tree. Compared with CHORD, cost of our system is only slightly higher. This is because the height of our tree could be as much as $1.44logN$, whereas for CHORD there is no such 1.44 factor. However, our system can support range queries efficiently while CHORD cannot.
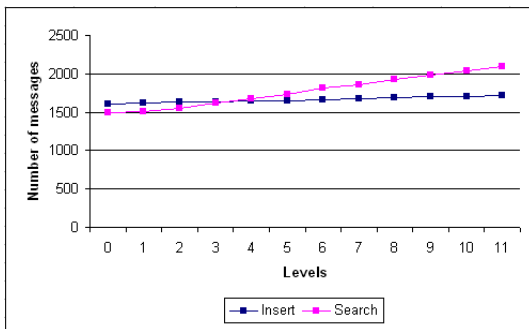
5.3 Access Load

Figure 10: Access load for nodes at different levels

Figure 10 shows access load of the nodes at different levels, measured in terms of the number of messages. For insertions, we find the load to be almost a constant across levels. For search, the load is slightly higher at the leaves than at the root, amply establishing that BATON does not overload nodes near the root of the network.

5.4 Effect of Load Balancing

To evaluate the capability of system in case of skewed data distribution, we test the network with a skewed data set, generated by Zipfian method with parameter 1.0. The result shows that there is no significant difference between costs of operations except that the load balancing process is triggered more frequently than that of uniform distributed data. Figure 11(a) shows
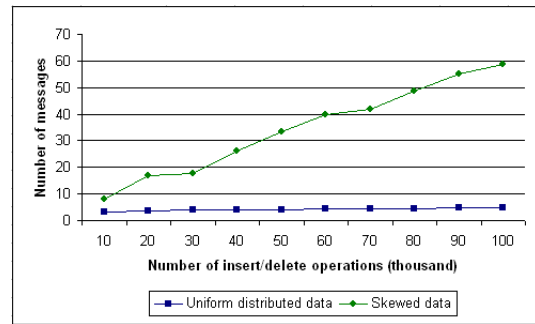
(a) Average messages of load balancing operation
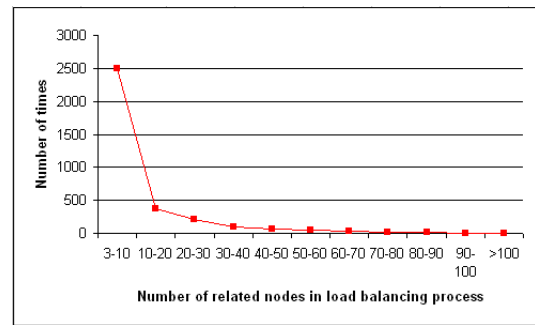
(b) Size of load balancing process

Figure 11: Load balancing operation

average number of messages required to balance the system in respective case of uniformly distributed data and skewed data. For skewed data, we find the cost of load balancing to grow linearly with the number of insert/delete's, with the expected number of load balancing messages per insertion/deletion to be about 1 message for every 1500 insertion/deletions; a very low overhead indeed.

To further understand this low cost, we plot in Figure 11(b) a distribution of the number of nodes involved in the load balancing operation. That is, how far did one have to shift to perform the forced insertion/deletion. The result is strongly exponential, showing that very little shifting is required most of the time, though long shifts may be required occasionally.

5.5 Effect of Network Dynamics

In P2P systems, nodes may join and leave at the same time. The intensity of nodes joining and leaving will have an effect on the robustness of the network. This experiment shows average extra messages taken due to concurrent joining or leaving operations. This is because it takes some times for the network to update knowledge of joining or leaving nodes, and during that time messages may be forwarded to wrong destination. The result in Figure 12 shows that the more nodes join or leave at the same time, more additional messages are taken.
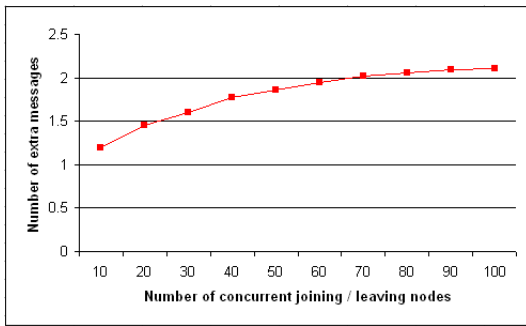
Figure 12: Network Dynamics

6 Conclusion

There is a plethora of overlay networks proposed for P2P systems. None of these are tree-structures in spite of tree structures being ubiquitous in data management. In this paper, we introduced BATON, a balanced binary tree overlay network for P2P systems.

By adding a small number of links in addition to the tree edges, we are able to obtain excellent fault tolerance, and also to get good load distribution without having to overload nodes near the root of the tree. We have shown how this tree structure can naturally be used to support an index structure for range queries. We have experimentally verified our complexity claims.

References

[1] K. Aberer. P-Grid: A self-organizing access structure for p2p information systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems*, 2001.

[2] J. Aspnes and G. Shah. Skip graphs. In *Proceeding of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, 2003.

[3] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-Trees. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 25–30, 2004.

[4] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of the 30th VLDB Conference*, 2004.

[5] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.

[6] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[7] D. Karger, F. Kaashoek, I. Stoica, R. Morris, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[8] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Professional, 1998.

[9] M. L. Lee, M. Kitsuregawa, B. C. Ooi, K.-L. Tan, and A. Mondal. Towards self-tuning data placement in parallel database systems. In *Proceedings of the 2000 ACM SIGMOD International Conference on the Management of Data*, pages 225–236, 2000.

[10] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th VLDB Conference*, pages 294–303, 1986.

[11] C. Y. Liau, W. S. Ng, Y. Shu, K.-L. Tan, and S. Bressan. Efficient range queries and fast lookup services for scalable p2p networks. In *Proceedings of 2nd International Workshop On Databases, Information Systems and Peer-to-Peer Computing*, pages 78–92, 2004.

[12] W. Litwin, M.-A. Neimat, and D. A. Schneider. Rp*: A family of order preserving scalable distributed data structures. In *Proceedings of the 20th VLDB Conference*, 1994.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable contentaddressable network. In *Proceedings of the 2001 ACM Annual Conference of the Special Interest Group on Data Communication*, pages 161–172, 2001.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference of Distributed Systems Platforms*, pages 329–350, 2001.

[15] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A peer-to-peer framework for caching range queries. In *Proceedings of the 20th International Conference on Data Engineering*, 2004.

[16] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report CSD-01-1141, Univ. California, Berkeley, CA, Apr. 2001.