

Self-Optimizing Distributed Trees

Michael K. Reiter

University of North Carolina
Chapel Hill, NC, USA
reiter@cs.unc.edu

Asad Samar

Goldman Sachs International
London, UK
Asad.Samar@gs.com

Chenxi Wang

Forrester Research
Foster City, CA, USA
chenxi.wang@gmail.com

Abstract

We present a novel protocol for restructuring a tree-based overlay network in response to the workload of the application running over it. Through low-cost restructuring operations, our protocol incrementally adapts the tree so as to bring nodes that tend to communicate with one another closer together in the tree. It achieves this while respecting degree bounds on nodes so that, e.g., no node degenerates into a “hub” for the overlay. Moreover, it limits restructuring to those parts of the tree over which communication takes place, avoiding restructuring other parts of the tree unnecessarily. We show via experiments on PlanetLab that our protocol can significantly reduce communication latencies in workloads dominated by clusters of communicating nodes.

1. Introduction

Tree structures are widely used in distributed applications including distributed directory protocols (e.g., [9, 30, 25]), peer-to-peer content sharing systems (e.g., [1, 31, 17, 19, 14]) and application-level multicast (e.g., [6, 15]), to name a few. In addition, several distributed implementations of important primitives such as mutual exclusion (e.g., [22, 5, 13, 28, 20]), and binary search (e.g., [11, 16, 21]) are built using tree structures, and these primitives are employed in a large number of distributed applications. The use of trees in different applications is warranted due to requirements specific to each setting, e.g., some applications map naturally to the tree hierarchy [17], others require the acyclicity of trees for simple protocol design [22, 5], and yet others utilize the locality-preserving properties of trees [31]. In all of these applications, a node in the tree communicates with other nodes using messages sent through the tree. Therefore, the worst case performance of these applications is proportional to the diameter (longest path between two nodes) of the tree. The trivial solution that makes the tree

“flat” (every node is a child of the root) does not scale well—the root becomes a bottleneck. Therefore, all such applications can benefit from a mechanism that would restructure the tree to reduce the number of communication hops required for a node to access another node (e.g., by reducing the diameter of the tree), while keeping a low fixed degree.

In this paper we present a distributed algorithm, called *flattening*, that improves the performance of accessing one node from another node in a k -ary tree (where each node has at most k children). Flattening achieves this through a novel restructuring technique that adjusts the tree as nodes communicate with each other, so as to optimize the tree according to the workload.

This restructuring has three properties. First, flattening brings nodes frequently accessing each other, closer to each other in the tree. Workloads in several applications are known to exhibit locality, in the sense that nodes that have communicated in the past are likely to communicate again in the future; such applications can benefit greatly from flattening. Note that in a degree-constrained tree (e.g., a k -ary tree), optimizing the access between a pair of nodes (by bringing them close to one another), could conflict with optimizing for another pair of nodes. This situation is further complicated due to the distributed nature of our algorithm: we assume that each node knows and communicates with only its neighbors in the tree, and has no information about the remaining tree topology. Flattening employs a distributed algorithm that utilizes only local information at each node, and finds a balance among conflicting optimization decisions by restructuring for a particular pair of nodes while at the same time preserving the effects of recent restructuring decisions made for other pairs.

Second, flattening has a tendency to reduce the diameter of the tree, without ever explicitly balancing the tree. In particular, it reduces the diameter of the component of the tree that spans nodes involved in recent accesses; note that this component may also contain nodes that are not involved in these accesses. Therefore, if the workload shows

no locality—e.g., if each node accesses a node chosen uniformly at random from all nodes in the tree—then flattening reduces the diameter of the whole tree, since in this case the component containing frequently accessed nodes would span most of the tree.

Finally, the restructuring steps are all local, i.e., each restructuring step at a node involves either only direct neighbors or at most neighbors of neighbors (nodes two hops away from each other) in the tree. This allows simple implementation of local policies at each node, e.g., a subtree containing nodes geographically close to each other could enforce a policy that prevents a geographically distant node from entering this subtree, as the tree is restructured. Furthermore, applications where nodes use a routing protocol to route messages to other nodes in the tree, benefit from the local restructuring steps, as nodes can easily update the routing tables according to the new topology.

We prove that flattening incurs a worst-case $O(\log n)$ amortized cost per flattening operation, where n is the number of nodes in the tree. Since the cost of this restructuring is directly tied to the cost of accessing another node—restructuring is performed along the path between the two nodes—the worst case cost of node accesses closely follows the $O(\log n)$ amortized performance of restructuring. Here we report empirical results from tests performed on Planet-Lab [7] that validate this analysis. We further implemented a flood-based access mechanism that runs on a tree, and allows nodes to access other nodes in the tree. We present results that demonstrate the performance of this flood-based protocol using our self-optimizing tree, and compare them to those obtained by running the same protocol on a randomly generated static tree over the same set of nodes. The flood-based protocol shows significant performance gains when utilizing the flattening algorithm.

We emphasize that our contribution lies in a tree optimization algorithm, not in implementing a full-blown application overlay. For example, we do not address several issues that a complete overlay solution would, including implementing various types of object queries (e.g., range queries); balancing application load; or dealing with failures of various types. Our primitive, however, complements tree-based technologies that address such issues (e.g., [1, 19]).

2. Related work

Our work is inspired by work on balancing binary search trees (BSTs) in a centralized system (e.g., [2, 26, 4, 12]), particularly the work on *splay trees* [26]. A splay tree is an elegant BST that achieves $O(\log n)$ amortized cost per access. When a node in the tree is accessed, splaying brings the node to the root of the tree, while balancing the tree in the process.

Our work differs from splaying in two ways. First, flattening employs a different heuristic that brings an accessed node close to the node initiating the access, by restructuring along the path between these two nodes—as opposed to bringing the accessed node close to the root by restructuring along the path to the root. In this way, our heuristic often enables more efficient implementations involving less restructuring than splaying. Indeed, the amount of restructuring performed by splay trees is a limitation in the centralized setting as well, and has been addressed previously; e.g., variants like *semi-splaying* [26], *randomized splaying* [3, 10] and *periodic splaying* [29], all attempt to reduce restructuring. We compare the restructuring costs of flattening versus splaying and its variants in our companion document [24].

Second, splaying uses non-local restructuring steps; this is mainly a result of restructuring in the context of binary search trees that require preserving the order of nodes in the tree. In particular, top-down splaying requires distant nodes in the original tree to form an edge with each other within a single restructuring step. This not only complicates enforcing local policies (like preserving geographic locality in the tree), but can also make it difficult for an application’s routing protocol to adjust routes according to the restructuring. Flattening improves on both of these aspects through local restructuring steps—an optimization enabled since our target setting does not require preserving the order of nodes in the tree.

Apart from splay trees, other balanced tree structures (e.g., [4, 18, 2, 12]) have also been proposed. Most of these proposals explicitly balance the tree with each insertion and deletion, incurring a high cost for these operations. Distributed implementations of some of these algorithms (but not splaying) have also been proposed (e.g., [11, 16, 21, 14]). Our approach is different in that it is less sensitive to insertions and deletions of nodes, and more sensitive to the actual workload. That is, our algorithm focuses its restructuring on the communication paths that are actually used, thereby yielding better performance for some workloads than even explicit balancing can achieve.

3. System model

Our system consists of n nodes distributed across a network and initially structured as a rooted, binary, unordered (and not necessarily complete) tree. (We will relax the assumption of a binary tree in Section 5.5.) Our algorithms make no assumptions about nodes joining or leaving the tree, except that the tree remains connected. Each node is initialized only with the identities of its neighbors in the tree, i.e., a parent pointer (the distinguished value “ \perp ” in the case of the root) and a set of child pointers of cardinality at most two. There is no central database accessible to nodes

that contains information about the tree structure. Nodes communicate via remote procedure calls (RPCs). Nodes and communication between them are reliable but asynchronous: nodes do not fail, and each RPC completes in a finite but unbounded time.

Nodes access other nodes according to application-specific protocols. We do not assume anything about these protocols except that two nodes in the tree communicate across the unique path between them in the tree, and so the access performance can be improved if this path contains fewer hops. The node that initiates the access request is denoted as the *requester*, and the node that is being accessed is denoted as the *target*.

4. Overview

At a high level, our algorithm works as follows: When a requester node r accesses a target node t , flattening is performed along the path between t and r . In particular, *bottom-up flattening* (Section 5.1) is employed while moving up the tree and *top-down semi-flattening* (Section 5.2) is used while moving down the tree. When this restructuring completes, t and r are closer to each other than before, and the height (distance from the root) of all the nodes in the path between t and r is reduced, i.e., the smallest subtree containing both t and r is left more balanced. This restructuring of the tree is not performed in the “critical path” of the access protocol, but rather as a background process: our tree simply “observes” the workload, and then optimizes itself so future accesses may be performed more efficiently.

In order to avoid concurrent restructuring of the tree, which would require expensive locking of parts of the tree, nodes implement mutually exclusive access to a shared *token*, i.e., there is only one “owner” of this token at a time. When a requester completes its access operation, it retrieves this token from the token’s previous owner, becoming the new owner itself. After becoming the new owner of the token, r notifies t , and t in turn initiates restructuring along the path to r . Our restructuring algorithm is not dependent on a particular protocol for managing mutually exclusive access to this token, though we do require that it enables nodes to navigate to the current owner of the token, i.e., allows a node x in the path from t to r to find the next node in this path (denoted as $\text{nextNode}(x, t, r)$ in our pseudo-code). Moreover, it must enable x to find if x is the highest node in this path or not (denoted as $\text{amHighNode}(x, t, r)$ in the pseudo-code). An example of such a protocol is Quiver [23], which maintains this navigation information through the use of “pointers” at each node that point to one of the neighbors of this node. These pointers need to be adjusted as flattening restructures the tree. Since flattening works in local restructuring steps, adjusting these pointers to reflect the new tree topology is feasible. Here we omit

the details of the Quiver protocol and the adjustments required to its state as the tree is restructured, in the interest of space; interested readers are referred to our companion documents [23, 24] for these details. We summarize the high-level steps as follows:

1. Requester r accesses target t through application-specific mechanisms.
2. r retrieves the token using the Quiver protocol, becoming the new (and only) owner of the token.
3. r notifies t , and t initiates flattening, with each node on the path from t to r using Quiver to navigate toward r .
4. Bottom-up flattening and top-down semi-flattening algorithms are used when going up and down the tree, respectively, along the path from t to r .

Note that steps 2–4 are not in the critical path of the application workload (step 1), and in particular, r (or any other node) may initiate subsequent access operations without waiting for the completion of steps 2–4.

5. Flattening algorithms

Binary search trees use the “move to front” heuristic and *rotate* accessed nodes close to the root, since all searches in a BST start from the root. In our setting however, a request may originate from any node in the tree. So a better heuristic is to move the targets close to the requesters. In order to achieve this, we rotate the requesters and the targets close to the root of the smallest subtree that contains them. This scheme minimizes restructuring in the tree (compared to some BSTs that rotate nodes all the way to the root of the tree) if the requester and the target are already close to each other. We implement this scheme by restructuring along the path from the target to the requester, employing both *bottom-up flattening* and *top-down semi-flattening* techniques, which we detail in Sections 5.1 and 5.2, respectively. This combination of “full” and “semi” flattening also allows our algorithm to adapt rather quickly to changing workloads while still being conservative about the number of messages exchanged for restructuring purposes.

Most existing restructuring techniques (again used in the context of BSTs) employ *rotation* as the basic restructuring step. This is convenient as rotation preserves the order of nodes in the tree—a requirement for binary search trees. Since ordering of nodes is irrelevant in our target protocols, we define and use new primitives that are better suited to our goals. Here we present these primitives and the bottom-up, top-down and hybrid (that combines bottom-up and top-down) flattening algorithms that use these primitives.

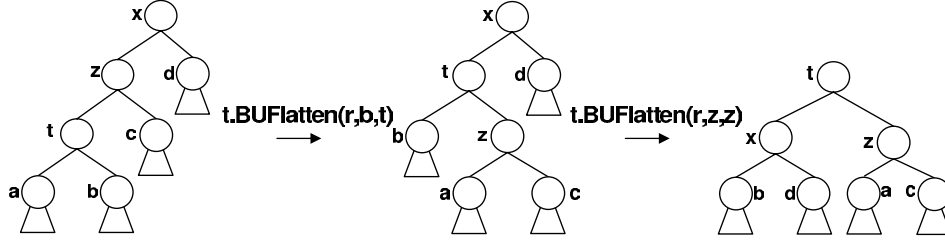


Figure 1. Bottom-up flattening: t first rotates over z and then x . Any child may be preferred for the first rotation (b preferred here). For subsequent rotations, preferred child is the one that t last rotated over (z here).

5.1. Bottom-up flattening

Our first algorithm is a bottom-up scheme that is employed when navigating up the tree from the target t to the requester r . Bottom-up flattening starts from t and proceeds to the highest node in the path to r . In case t is this highest node, no bottom-up restructuring is performed. The result of bottom-up flattening is to bring t to the root of the subtree that contains r , while leaving the subtree containing t and r more balanced than before.

5.1.1. Preferred rotation primitive

We define a variation of the well-known rotation primitive, for bottom-up flattening. For each rotation performed by the target t over its parent z , t chooses one of its children as a *preferred child*. The rotation is performed such that t keeps the preferred child and “hands off” the other child to z . We call this a *preferred rotation*. Preferred rotations are used in bottom-up flattening as shown in Figure 1. For the first rotation, t chooses either one of its children as the preferred child. For each subsequent rotation, the child that t just rotated over in the previous step (node z in Figure 1) is preferred. t performs these steps until it rotates over the highest node in the path to r .

5.1.2. Bottom-up flattening algorithm

Figure 2 shows the distributed algorithm that implements bottom-up flattening. We denote the variables encoding persistent state at a node y using the prefix “ y .”, e.g., y .parent. Variable names without the prefix denote temporary state that is deleted once this invocation is over.

The target t initiates bottom-up flattening by invoking t .BUFlatten(r, b, t), where r is the requester and b is t ’s preferred child. In line 2, $\text{elmt}(S)$ simply returns the element of a singleton set S ; this element is the non-preferred child of t . If t is initially a leaf node then $b = \perp$ and $a = \perp$ (line 2). If t only has one child then b is that child and

$a = \perp$. We assume that when there is a remote invocation on a \perp node, the method returns (possibly with an error message) so the invoking node can carry on its execution. The rotateEdge invocation (line 5) results in z setting z .parent to t (line 13) and adding t ’s non-preferred child a to z .children, replacing t (line 14). It then returns its previous parent x and whether z was the highest node in the original path from t to r before restructuring began (line 15). Note that t ’s new parent after each preferred rotation (t ’s grand-parent before the rotation) need not be notified of its new child t , since t is going to rotate over this node anyway in the next step. Therefore, at each subsequent step after the first rotation, t .parent does not contain t in its children set but rather contains the node z that t just rotated over in the previous step. After the last rotation, t .parent is notified of its new child (line 9). The RPCs in lines 5, 7 and 9 ensure that all restructuring is complete by the time the last rotation completes.

5.2. Top-down semi-flattening

Our second algorithm is a top-down scheme that restructures the tree when navigating down the tree from t to r . Top-down semi-flattening starts at the highest node in the path from t to r and brings r part way up to this highest node. If top-down semi-flattening is preceded by the bottom-up variant (as in hybrid flattening, Section 5.3), this highest node is, in fact, t .

5.2.1. Child swap primitive

Top-down semi-flattening is performed by repeating the step shown in Figure 3. y , x and a are in the path from t to r . Node y swaps its child c with x ’s child a . We call this step *child swap*. “+” represents the current node of the flattening operation, i.e., the next child swap is performed by a . Top-down semi-flattening is initiated by the highest node in the path between t and r , and terminates if r is the current node or a child of the current node.

```

1.  $t.BUFlatten(r, b, w)$                                 /*  $r$ : requester,  $b$ : preferred child,  $w$ : former child of  $t.parent$  */
2.    $a \leftarrow \text{elmt}(t.children \setminus \{b\})$       /*  $a$  is the child not preferred */
3.    $z \leftarrow t.parent$                                /*  $z$  is the current parent */
4.    $t.children \leftarrow \{t.children \setminus \{a\}\} \cup \{z\}$  /* replace child  $a$  with  $z$  */
5.    $[gParent, isHigh] \leftarrow z.rotateEdge(t, r, w, a)$  /*  $z$  replaces its child  $w$  with  $a$  and sets  $z.parent$  to  $t$  */
6.    $t.parent \leftarrow gParent$                          /* set new parent to old grand-parent */
7.    $a.setParent(z)$                                      /*  $a.parent$  now points to  $z$  */
8.   if  $isHigh$  is true                                 /* if  $z$  was the highest node in the path, then... */
9.      $t.parent.replaceChild(z, t)$                      /* ...my new parent replaces its child  $z$  with me and stop */
10.  else  $t.BUFlatten(r, z, z)$                          /* otherwise, perform next rotation preferring  $z$  */

11.  $z.rotateEdge(t, r, w, a)$                            /*  $t$ : target,  $r$ : requester,  $w$ : child to replace,  $a$ : new child */
12.   $x \leftarrow z.parent$                                /*  $x$  is my current parent */
13.   $z.parent \leftarrow t$                                /* set  $t$  as new parent */
14.   $z.children \leftarrow \{z.children \setminus \{w\}\} \cup \{a\}$  /* replace child  $w$  with  $a$  */
15.  return  $[x, \text{amHighNode}(z, t, r)]$                 /* return  $x$  and if I am the highest node in this path or not */

16.  $x.replaceChild(z, t)$                                /*  $z$ : child to replace,  $t$ : new child */
17.   $x.children \leftarrow \{x.children \setminus \{z\}\} \cup \{t\}$  /* replace child  $z$  with  $t$  and return */

18.  $a.setParent(z)$                                      /*  $z$ : new parent */
19.   $a.parent \leftarrow z$                                /* set parent to  $z$  and return */

```

Figure 2. Bottom-up flattening. All nodes implement all algorithms.

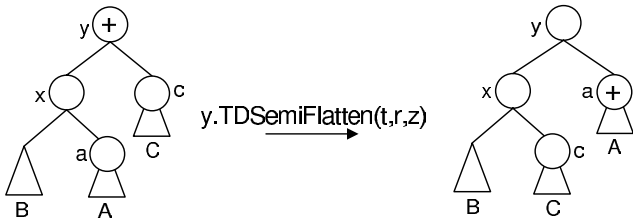


Figure 3. Top-down semi-flattening: y , x and a are in the path from t to r . z is y 's parent (not shown). Next invocation is $a.TDSemiFlatten(t, r, y)$.

5.2.2. Top-down semi-flattening algorithm

Figure 4 shows the distributed algorithm for this scheme. The algorithm is initiated by the highest node h in the path from t to r as $h.TDSemiFlatten(t, r, h.parent)$. At each step, the current node y and its child x on the path from t to r swap y 's child that is not in this path with x 's child that is in this path (lines 9 and 13). The children are notified of their new parents (lines 7, 10).

Top-down semi-flattening approximately halves the depth of each node (relative to h) in the path from h to r ;

again note that $h = t$ in case bottom-up flattening is employed before top-down semi-flattening. As a result, semi-flattening brings r closer to t .

5.3. Hybrid flattening

Our main algorithm combines bottom-up flattening with top-down semi-flattening to restructure along the path from the target t to the requester r . Figure 5 shows the distributed algorithm for hybrid flattening. t performs bottom-up flattening if it is not the highest node (lines 2–7). This results in t becoming the root of the subtree that contains r . After bottom-up flattening is complete, if r is a child of t then no more restructuring is required (line 8). Otherwise, t initiates top-down semi-flattening (line 9) that continues until r is reached.

Figure 6 shows an example tree where flattening is performed between t and r . Hybrid flattening brings t and r close to each other and in the process, balances the subtree containing t and r .

5.4. Analysis

Flattening minimizes the number of messages exchanged during each flattening step, while maximizing the effects of

```

1.  $y.$ TDSemiFlatten( $t, r, z$ )           /*  $t$ : target,  $r$ : requester,  $z$ : my new parent */
2.    $y.$ parent  $\leftarrow z$ 
3.   if  $r \in \{y\} \cup y.$ children       /* if I or my child is the requester, then...*/
4.     stop                             /* ...stop the restructuring */
5.    $x \leftarrow \text{nextNode}(y, t, r)$    /* find the child that is in path from  $t$  to  $r$  */
6.    $c \leftarrow \text{elmt}(y.$ children  $\setminus \{x\}$ ) /* this is the child not in path */
7.    $c.$ setParent( $x$ )                   /*  $c$ 's parent should now be  $x$  */
8.    $a \leftarrow x.$ childSwap( $t, r, c$ ) /* swap children at  $x$  and get  $a$ , the grand-child in path */
9.    $y.$ children  $\leftarrow \{y.$ children  $\setminus \{c\}\} \cup \{a\}$  /* swap child with grand-child */
10.   $a.$ TDSemiFlatten( $t, r, y$ )         /* initiate next child swap; this RPC can be non-blocking */

11.  $x.$ childSwap( $t, r, c$ )             /*  $t$ : target,  $r$ : requester,  $c$ : my parent's child not in path */
12.   $a \leftarrow \text{nextNode}(x, t, r)$    /* find my child that is in path from  $t$  to  $r$  */
13.   $x.$ children  $\leftarrow \{x.$ children  $\setminus \{a\}\} \cup \{c\}$  /* swap child with parent's child */
14.  return  $a$                        /* return my child that has been swapped */

```

Figure 4. Top-down semi-flattening. All nodes implement all algorithms.

```

1.  $t.$ HybridFlatten( $r$ )                 /*  $r$ : requester */
2.   if amHighNode( $t, t, r$ ) is false /* BUFlatten if I am not the highest node */
3.      $\{a, b\} \leftarrow t.$ children   /*  $a$  and  $b$  are  $t$ 's children, could be null */
4.     if  $a = \perp$ 
5.       prefChild  $\leftarrow b$        /* choose the non-null child as the preferred child */
6.     else prefChild  $\leftarrow a$      /* if both are null or both are non-null then choose any */
7.      $t.$ BUFlatten( $r, \text{prefChild}, t$ ) /* do bottom-up flattening */
8.   if  $r \notin t.$ children           /* if more than one hop away from  $r$ , then... */
9.      $t.$ TDSemiFlatten( $t, r, t.$ parent) /* ...do top-down semi-flattening */

```

Figure 5. Hybrid flattening algorithm.

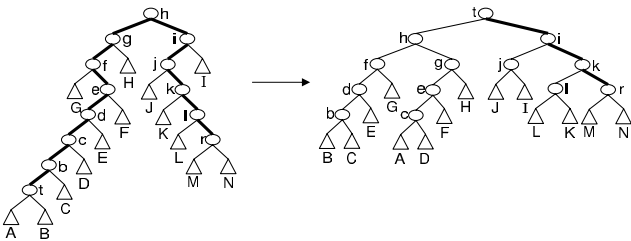


Figure 6. Hybrid flattening. Bold lines show the path between t and r . Root of A was the first preferred child.

the restructuring in balancing the tree and bringing the requesters and their targets closer to each other. In particular, each preferred rotation (used in bottom-up flattening)

requires only 4 messages—two messages for each of the two RPCs in lines 5 and 7 in Figure 2—except for the last rotation that requires 6 messages due to line 9 in Figure 2. Each child swap (used in top-down semi-flattening) requires 5 messages and moves two steps down the tree—two messages for each of the two RPCs in lines 7 and 8 in Figure 4 and an additional message for the RPC in line 10 in Figure 4. (This RPC may be non-blocking, and so we do not count its response against the latency of the child swap.)

We perform a detailed analysis of the amortized cost of flattening using the potential method [27]. We assign a real number called *potential* to each possible state of the tree. A *potential function* is a mapping from the tree states to the potential. The *expense* of an operation in the potential method is defined as the sum of the actual work of the operation and the net increase in the potential as a result of this operation. Using this definition, the total actual work of a

sequence of m operations can be derived as:

$$\text{total actual work} = \text{total expense} + \text{net decrease in potential} \quad (1)$$

Our proof strategy to bound the total actual work of a sequence of operations is to bound the expense of the sequence of operations (Lemma 1 for top-down and Lemma 2 for bottom-up flattening) and the net decrease in potential (Lemma 3) resulting from the sequence of operations. For this proof, we assume that the tree contains a static set V of n nodes. Note that this is only required to quantify the cost of flattening in terms of the size of the tree; our algorithm does not require a static set of nodes or an upper bound on the number of nodes in this set.

We begin by assigning a positive weight $w(x)$ to each node x that remains fixed throughout the execution. Then define the size $s(x)$ of a node x to be the sum of weights of all nodes in the subtree rooted at x . We define the rank $r(x)$ of x as $\log(s(x))$ (binary logarithms are used throughout). The potential function is just the sum of the ranks of all nodes in the tree. As a measure of the actual work, we charge one work unit for each child swap and preferred rotation. Since each child swap and preferred rotation is performed with a fixed number of RPCs (described above), our analysis with work unit one suffices to yield an asymptotic bound, in that the constants can be immediately derived from the discussion above on the number of messages required in each step. We use s and s' , r and r' to denote the sizes and ranks of nodes just before and after a restructuring step, respectively.

Lemma 1. *The expense of top-down semi-flattening from a node t to a node r is at most $2(r(t) - r(r))$.*

Proof. Top-down semi-flattening consists of child swaps. The expense of top-down semi-flattening is the sum of the expense of all the child swaps from t to r . We claim that the expense of a single child swap with x being the parent of a and y being the parent of x (see Figure 3) is at most $2(r(y) - r'(a))$. The sum of these child swap expenses “telescopes” to $2(r(t) - r(r))$ if the path length between t and r is even and $2(r(t) - r(r'))$ if this length is odd, where r' is the parent of r . The Lemma holds in either case since $r(r') \geq r(r)$.

So we only need to prove the claim regarding the expense of each child swap. The child swap is as shown in Figure 3. The actual number of work units associated with a child swap is one so the expense is:

$$\begin{aligned} & 1 + \text{net increase in potential} \\ &= 1 + r'(x) - r(x) \quad [\text{since only } x\text{'s rank changes}] \\ &\leq 1 + r'(x) - r(a) \quad [\text{since } r(x) \geq r(a)] \end{aligned}$$

Now we need to prove that $1 + r'(x) - r(a) \leq 2(r(y) - r'(a))$, which we rearrange as follows, with each line being

equivalent:

$$\begin{aligned} & 1 \leq 2r(y) - 2r'(a) + r(a) - r'(x) \\ & 1 \leq 2r(y) - r'(a) - r'(x) \quad [\text{since } r'(a) = r(a)] \\ & -1 \geq r'(a) - r(y) + r'(x) - r(y) \\ & -1 \geq \log\left(\frac{s'(a)}{s(y)}\right) + \log\left(\frac{s'(x)}{s(y)}\right) \end{aligned}$$

This is true since $s(y) \geq s'(a) + s'(x)$ and $\log b + \log c$ maximizes at -2 if $b + c \leq 1$ (convexity of \log). \square

Lemma 2. *The expense of bottom-up flattening from a node t to a node h is at most $2(r(h) - r(t)) + 1$.*

Proof. Bottom-up flattening consists only of preferred rotations. To see the effects of preferred rotations on the expense of bottom-up flattening, we need to analyze two preferred rotations at a time. Bottom-up flattening consists of these pairs of preferred rotations, possibly followed by a single preferred rotation at the end, in case the path between t and h is of odd length.

Let z be the parent of t and x be the parent of z as shown in Figure 1. t is the node that performs the preferred rotations. We claim that the expense of a single preferred rotation is at most $2(r'(t) - r(t)) + 1$ and that of a pair of preferred rotations is at most $2(r'(t) - r(t))$. The sum of these expenses telescopes and proves the lemma. We now prove our claim.

The actual number of work units of a preferred rotation performed by t over z is one. The expense is:

$$\begin{aligned} & 1 + r'(t) - r(t) + r'(z) - r(z) \\ & \leq 1 + r'(t) - r(t) \quad [\text{since } r'(z) \leq r(z)] \\ & \leq 1 + 2(r'(t) - r(t)) \quad [\text{since } r'(t) \geq r(t)] \end{aligned}$$

The actual number of work units of a pair of preferred rotations performed by t over z and then over x (see Figure 1) is two. The amortized expense is:

$$\begin{aligned} & 2 + r'(t) - r(t) + r'(z) - r(z) + r'(x) - r(x) \\ &= 2 - r(t) + r'(z) - r(z) + r'(x) \quad [\text{since } r'(t) = r(x)] \\ &\leq 2 + r'(z) + r'(x) - 2r(t) \quad [\text{since } r(t) \leq r(z)] \end{aligned}$$

Now we need to prove that $2 + r'(z) + r'(x) - 2r(t) \leq 2(r'(t) - r(t))$, which we rearrange as follows with each line being equivalent:

$$\begin{aligned} & 2 \leq 2r'(t) - r'(z) - r'(x) \\ & -2 \geq r'(z) - r'(t) + r'(x) - r'(t) \\ & -2 \geq \log\left(\frac{s'(z)}{s'(t)}\right) + \log\left(\frac{s'(x)}{s'(t)}\right) \end{aligned}$$

This is true since $s'(t) \geq s'(z) + s'(x)$ and $\log b + \log c$ maximizes at -2 if $b + c \leq 1$ (convexity of \log). \square

Lemma 3. *The net decrease in potential over any sequence of operations is at most $\sum_{v \in V} \log(\frac{W}{w(v)})$, where $W = \sum_{v \in V} w(v)$.*

Proof. The maximum size of a node v , for all $v \in V$, is W when v is the root of the tree and the minimum size is $w(v)$ when v is a leaf. Thus the net decrease in the rank of node v is at most $\log(W) - \log(w(v))$. Summing up over all nodes proves the lemma. \square

Theorem 1. *The total actual work done by a sequence of m top-down flattening operations is at most $(2m + n) \log n$.*

Proof. Assign a weight of $1/n$ to each node, and so $W = 1$. The total expense of the sequence is at most $m(2(\mathbf{r}(t) - \mathbf{r}(r))) \leq 2m \log n$ for any t and r , see Lemma 1. The net decrease in potential is at most $\sum_{v \in V} \log(\frac{W}{w(v)}) = n \log n$. Substituting these values in Equation 1 proves the result. \square

Theorem 2. *The total actual work done by a sequence of m bottom-up flattening operations is at most $m + (2m + n) \log n$.*

Proof. Assign a weight of $1/n$ to each node. The total expense of the sequence is at most $m(1 + 2(\mathbf{r}(h) - \mathbf{r}(t))) \leq m + 2m \log n$ for any h and t , see Lemma 2. The net decrease in potential is at most $\sum_{v \in V} \log(\frac{W}{w(v)}) = n \log n$. Substituting these values in Equation 1 proves the result. \square

Theorem 3. *The total actual work done by a sequence of m hybrid flattening operations is at most $3m + (2m + n) \log n$.*

Proof. Assign a weight of $1/n$ to each node. The total expense of the sequence is at most $m(1 + 2(\mathbf{r}(h) - \mathbf{r}(t)) + 2(\mathbf{r}''(t) - \mathbf{r}(r)))$ for any t, h and r , where $\mathbf{r}''(t)$ is the rank of t after bottom-up flattening, see Lemmas 1 and 2. Note that $\mathbf{r}''(t)$ is the same as the rank of h before bottom-up flattening (the subtree contains the same nodes), so the total expense of the sequence is at most $m(1 + 2(\mathbf{r}(h) - \mathbf{r}(t)) + 2(\mathbf{r}(h) - \mathbf{r}(r))) \leq m + 2m \log 2n = 3m + 2m \log n$ for any t, h and r . The net decrease in potential is at most $\sum_{v \in V} \log(\frac{W}{w(v)}) = n \log n$. Substituting these values in Equation 1 proves the result. \square

A consequence of these results is that as m grows, our restructuring induces an average path length between nodes that access one another of $O(\log n)$ in size. If access patterns are stable, they will inherit the benefits of conducting accesses over these short paths. We confirm this intuition empirically in Section 6.

5.5. K-ary trees

Our algorithms as described in the previous sections work only for a binary tree. However, extension to k -ary trees is straightforward. In both bottom-up flattening and top-down semi-flattening, each step consists of a node replacing one of its children—let us denote this as the *least significant node*—with a node in the path to the requester—denote it as the *most significant node*. In the first step of Figure 1, the root of subtree A is the least significant node and z is the most significant node whereas in Figure 3, c is the least significant node and a is the most significant node. In the case of a k -ary tree, the most significant node is still well-defined (the node in the path to the requester) but the least significant node is not. A simple strategy to define the least significant node could be the following: If a node x in a k -ary tree has $k' < k$ children, then we say it has $k - k'$ null children. x prioritizes its children according to some heuristic, e.g., a least recently used (LRU) type algorithm that gives a higher priority to a child that was in the access path of the most recent access through x . The null children always get the lowest priority. Then x may choose the child with the lowest priority as the least significant node when restructuring.

6. Experiments

We have completed an implementation of the flattening algorithms described in the previous sections including an implementation of the Quiver protocol [23] for mutually exclusive access to the token, and for navigating along the path from the target to the requester (see Section 4).

6.1. Experimental setup

Our experiments were run on PlanetLab [8] using 40 nodes spread across North America. For each experiment, these 40 nodes were arranged in a binary tree; we chose a binary tree (versus a k -ary tree for $k > 2$) so as to maximize the tree diameters experienced with only 40 nodes. After initializing each node with information about its parent and children, nodes initiated an application workload. In order to emulate an application that uses the tree structure for communication between nodes, each node performed flood-based searches for other nodes in the tree. In each of these searches, the requester node broadcast a packet to each of its neighbors with the identity of the target node. Each node (except the target) that received such a packet, forwarded it to each of its neighbors, except the neighbor from where the packet came. When the target node itself received this packet, it sent an acknowledgment directly (outside the tree) to the requester. After receiving the acknowledgment from the target, the requester measured the latency of the access

and then retrieved the token and notified the target, which then initiated the flattening algorithm along the path to the requester.

In order to control the sequence of requests (so we could construct worst cases and other distributions), we used one node external to the tree as a “monitor”. The monitor exchanged control messages with all nodes, e.g., to have nodes initiate an access request or to pull information about how long an access operation took.

We performed three sets of experiments: the first employed a randomly constructed tree and a random workload, i.e., each node chose its target uniformly at random among all nodes. The second set again employed a randomly constructed tree, but used a workload where nodes were divided in groups such that nodes in a particular group accessed each other more frequently. The final set utilized a location-aware tree, i.e., geographically nearby nodes were placed close to each other in the tree, and a random workload. Each data point in our graphs is an average value over four runs.

6.2. Random tree, random workload

For these tests we constructed a random binary tree among 40 PlanetLab nodes. In order to perform an access operation, a node chose another node uniformly at random from the set of all 39 other nodes in the tree, and initiated a flood-based access request for the chosen node. The top of Figure 6.2 presents the amortized latencies of the access operations with and without using flattening. Flattening significantly improved the performance of access operations, and it did so by reducing the diameter of the tree; see the bottom of Figure 6.2. This is further confirmed by comparing the tree topologies at the beginning and end of each experiment. An example is shown in Figure 6.2. Note that since nodes chose their targets at random, the set of recent requesters and targets span most of the tree, and so flattening has the effect of balancing the whole tree.

6.3. Random tree, group workload

In a second set of tests, we again arranged the 40 PlanetLab nodes in a random binary tree. In these tests, though, we partitioned the nodes into 10 non-intersecting groups of four nodes each. When selecting a node to access via the flooding algorithm, each node selected from within its group with probability 0.8, and selected from outside its group with probability 0.2. We hypothesized that with such a workload, our approach would tend to bring the members of each group closer to one another, thereby improving the latency of intra-group accesses.

The results from these tests are shown in Figure 9. This figure shows the average access latencies that resulted when

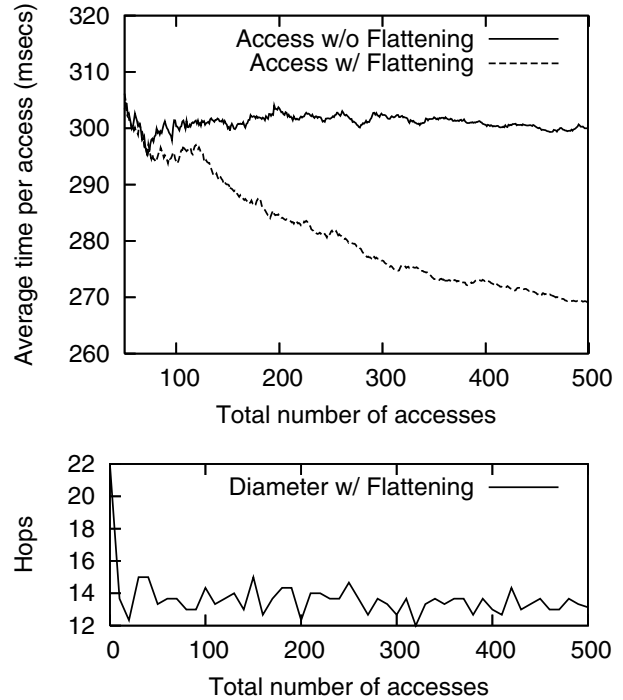


Figure 7. Random workload on a randomly constructed tree (Section 6.2). Top shows amortized access latencies with and without flattening. Bottom shows the diameter of the tree with flattening.

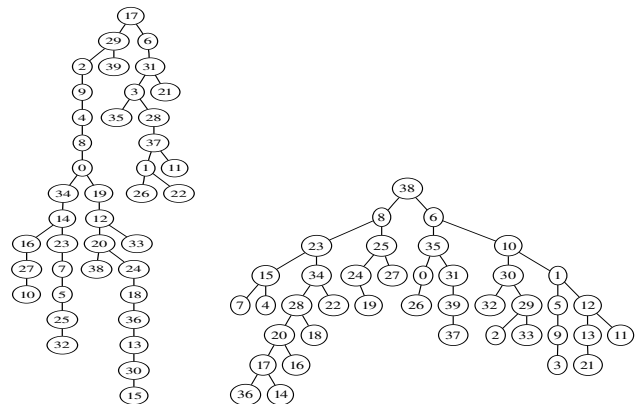


Figure 8. An example start (left) and end (right) topology from an experiment with a random workload on a randomly constructed tree (Section 6.2).

flattening was or was not used. As the top portion shows, the access latency was dramatically improved through the use

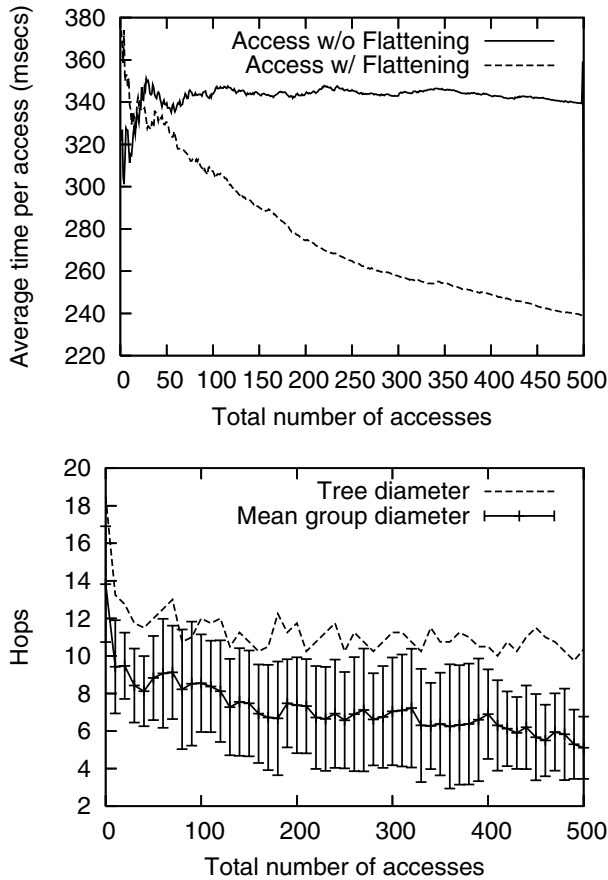


Figure 9. Latencies (top) and tree and group diameters (bottom) in group-biased workload (Section 6.3).

of flattening. The reason for this improvement is demonstrated in the bottom portion of the same figure, which shows the average tree diameter and the average group diameter, i.e., the hops between the furthest members in each group, averaged over all groups. The error bars in this bottom graph show the standard deviation of the group diameters. As this graph shows, the tree and group diameters drop rapidly at first, and then continue a slight downward trend for the duration of the workload. This, in turn, translates to significant latency savings for access (see top of Figure 9).

This conclusion is also supported by examining the topologies that resulted from our experiments. For example, Figure 10 shows an initial and an ending topology in one of our experiments. The label on each node indicates the group of which that node is a member. The right tree also shows the groups that end with the largest group diameter (in gray) and that end with the smallest group diameter (in black); these two groups are colored similarly in the left tree to show where these groups began in the initial topol-

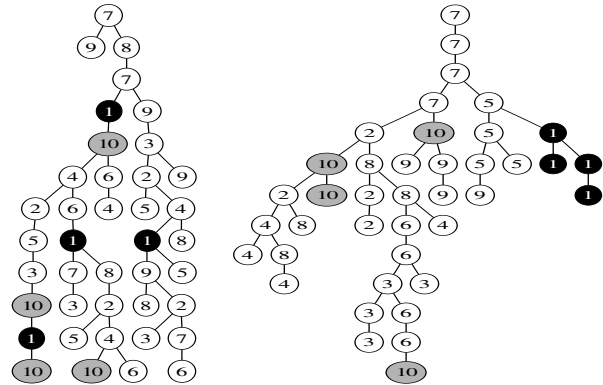


Figure 10. Example PlanetLab node topologies at the start (left) and end (right) of the group-bias experiments (Section 6.3). Circles with the same numbers represent nodes in the same group. The gray group (10) ends with the worst diameter, and the black group (1) ends with the best. Flattening reduces group diameters overall.

ogy. A careful examination of the various groups shows that the group diameters became smaller during the workload due to flattening.

6.4. Geographic tree, random workload

Our last experiments organized the 40 nodes geographically, by partitioning the nodes into “west coast”, “east coast” and “central” nodes; each such group occupied a contiguous portion of the tree initially. We then performed random workloads in which each node, to perform an access, selected a node to access uniformly at random from among the 39 other nodes. In this context we explored three different restructuring regimes: no restructuring; universal flattening without attention to the geographic partitioning; and geographic flattening, i.e., flattening within each geographic region only. That is, a path between a requester and a target that traversed multiple regions was restructured only on its contiguous subpaths within each region; connections between regions and the nodes they connected were left alone.

We explored this workload to highlight a feature of flattening, namely that each restructuring step, being localized to the vicinity of the node executing it, can be applied in subtrees effectively and enables the node to apply localized policies that limit restructuring. The benefit that this offers is convincingly demonstrated in Figure 11. As shown, the restructuring that respected local geographic policies outperformed both no restructuring and universal restructuring, and in fact the universal restructuring performed no better

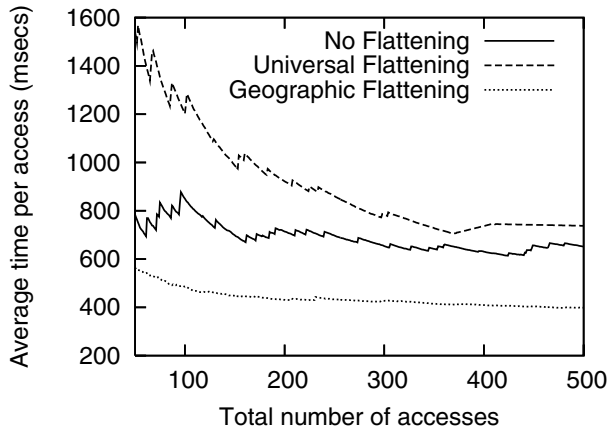


Figure 11. Access latencies in experiments beginning with a geographically placed tree and running a random workload with geographic flattening (Section 6.4).

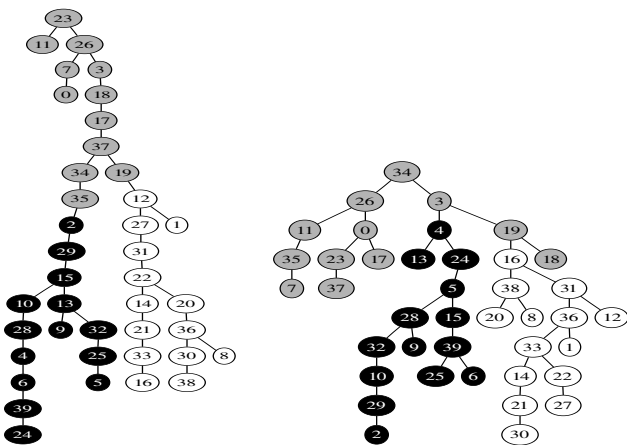


Figure 12. An example start (left) and end (right) topology from an experiment beginning with a geographically placed tree and running a random workload with geographic flattening (Section 6.4). Nodes are colored according to their regions.

than no restructuring due to its failure to account for geographic realities. An example topology produced by this geographic restructuring is shown in Figure 12.

7. Conclusions

In this paper, we presented a novel algorithm by which a distributed tree restructures itself dynamically to offer lower-latency communication to the applications that run

over it. Our restructuring method shortens the path between nodes that have communicated with each other in the past, and so these nodes will benefit if they communicate in the future (i.e., if the application exhibits communication locality). Moreover, it does so through a series of inexpensive, primitive steps performed by nodes on that path; each such step involves localized communications with its neighbors, simplifying the adaptation of the applications' routing data structures to reflect the restructuring. We have implemented our approach and confirmed the performance benefits it offers using experiments on PlanetLab.

8. Acknowledgments

This work was supported in part by NSF awards 0326472 and 0433540. We are grateful to the reviewers for their comments, which were useful for improving this paper.

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] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet Math. Dokl.* 3, pages 1259–1263, 1962.
- [3] S. Albers and M. Karpinski. Randomized splay trees: theoretical and experimental results. *Information Processing Letters*, 81(4):213–221, 2002.
- [4] R. Bayer. Symmetric binary B-Trees: data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.
- [5] Y. Chang, M. Singhal, and M. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Proc. 9th IEEE Symp. on Reliable Dist. Syst.*, pages 146–154, 1990.
- [6] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast*, 20(8), 2002.
- [7] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.
- [8] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.
- [9] M. J. Demmer and M. Herlihy. The Arrow distributed directory protocol. In *Proc. 12th Intl. Symposium of Distributed Computing*, pages 119–133, 1998.
- [10] M. Furer. Randomized splay trees. In *Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.
- [11] K. Gilon and D. Peleg. Compact deterministic distributed dictionaries. In *Proc. of the Annual ACM Symposium on Principles of Distributed Computing*, 1991.

- [12] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, 1978.
- [13] J. M. Helary, A. Mostefaoui, and M. Raynal. A general scheme for token- and tree-based distributed mutual exclusion algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 5(11):1185–1196, 1994.
- [14] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st International Conference on Very Large Databases*, Aug. 2005.
- [15] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O’Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 2000.
- [16] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the B-link tree. Technical Report MIT/LCS/TR-530, Massachusetts Institute of Technology, 1992.
- [17] R. Kurmanowysch, M. Jazayeri, and E. Kirda. Towards a hierarchical, semantic peer-to-peer topology. In *Proceedings of the 2nd International Conference on Peer-to-Peer Computing*, 2002.
- [18] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.
- [19] C. Y. Liao, 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 the 2nd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, pages 78–92, 2004.
- [20] M. Naimi, M. Trehel, and A. Arnold. A log (n) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, 1996.
- [21] D. Peleg. Distributed data structures: a complexity oriented view. In *Proc. 4th International Workshop on Distributed Algorithms*, pages 71–89, 1990.
- [22] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, Feb. 1989.
- [23] M. K. Reiter and A. Samar. Quiver: Consistent object sharing for edge services. *IEEE Transactions on Parallel and Distributed Systems*, 2007. To appear.
- [24] A. Samar. *Quiver on the Edge: Consistent, Scalable Edge Services*. PhD thesis, Carnegie Mellon University, Aug. 2006.
- [25] B. Silaghi, B. Bhattacharjee, and P. Keleher. Query routing in the TerraDir distributed directory. In *Scalability and Traffic Control in IP Networks II*, pages 299–309, July 2002.
- [26] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.
- [27] R. E. Tarjan. Amortized computational complexity. *SIAM J. Appl. Discrete Math*, 6:306–318, 1985.
- [28] S. Wang and S. Lang. A tree-based distributed algorithm for the k-entry critical section problem. In *IEEE International Conference on Parallel and Distributed Systems*, Dec. 1994.
- [29] H. E. Williams, J. Zobel, and S. Heinz. Self-adjusting trees in practice for large text collections. *Software, Practice and Experience*, 31(10):925–939, 2001.
- [30] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol, 1995. RFC 1777.
- [31] C. Zhang, A. Krishnamurthy, and R. Y. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems*, Feb. 2005.