# Self-Adjusting Distributed Trees

**Michael K. Reiter**[1] **Asad Samar**[2] **Chenxi Wang**[2]

September 2005
CMU-CS-05-171

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]Electrical & Computer Engineering Department and Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; reiter@cmu.edu

[2]Electrical & Computer Engineering Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA; {asamar,chenxi}@ece.cmu.edu

**Keywords:** Tree balancing, Distributed algorithms, Amortized cost

**Abstract**

An object retrieval protocol that enforces mutually exclusive access to a shared object is an important primitive employed by many distributed applications including distributed directories, distributed resource sharing systems and ordered multicast protocols, to name a few. Most existing implementations of this object retrieval primitive use a tree as the underlying communication structure due to the simple acyclic nature of trees. The worst case performance of this primitive and of the large body of applications built upon it, is $O(n)$ for $n$ nodes sharing the object. In this paper, we present a novel distributed self-adjusting tree for object retrieval protocols that guarantees the message complexity per retrieval, averaged over the worst case sequence of retrievals, to be $O(\log n)$. In addition, our algorithm adjusts only portions of the tree in which retrievals occur; this is advantageous when the tree structure reflects network proximity. We implement best known techniques from the centralized setting and compare their performance with our algorithm. Results are presented from experiments carried out on PlanetLab to evaluate the performance of different schemes under different workloads. We also present extensions to our basic protocol allowing a wide range of distributed applications including atomic broadcast and content discovery to achieve better performance using our techniques. To our knowledge, this is the first attempt to reduce access costs in a distributed tree where the tree dynamically adjusts itself during use to achieve $O(\log n)$ performance for worst case workloads.

# 1 Introduction

The hierarchical acyclic structure of trees allows the use of simple distributed algorithms for data sharing and coordination among nodes in the tree. An important primitive used in many of these algorithms is an object retrieval protocol that enforces mutually exclusive access to a shared mobile object. The object can be retrieved by a requesting node from the current owner. When the retrieval is complete, the requesting node becomes the new owner of the object. Many implementations of this primitive are known including several token-based distributed mutual exclusion solutions [26, 5, 12, 33, 23] and distributed directory protocols [7]. In all these cases, nodes communicate by exchanging messages that are routed across tree edges. Therefore, the worst case performance of these algorithms and of the large body of applications built upon these primitives [15, 13, 14] is proportional to the diameter (longest path between two nodes) of the tree. The trivial solution to make the tree "flat" (every node is a child of the root) does not scale well—the root becomes a bottleneck. Therefore, all such algorithms can benefit from a distributed mechanism that would reduce the diameter of the tree, e.g., by explicitly balancing or heuristically restructuring the tree, while keeping a low fixed degree.

In this paper we present a novel distributed algorithm, called *flattening*, that improves the worst case performance of the object retrieval protocols mentioned above. In particular, flattening achieves an *amortized cost*—cost per retrieval averaged over the worst case sequence of retrievals—of $O(\log n)$, where $n$ is the number of nodes in the tree. This is a result of employing a new restructuring heuristic that brings nodes frequently requesting the object closer to each other. Flattening achieves this amortized cost while keeping the cost of individual retrieval operations low. Flattening also has a tendency to preserve the structure of the tree, in that the nodes that are close to one another tend to stay close to one another if the workload permits. So, for example, if the tree edges connect nodes that are geographically close to one another, and if the object is passed among nodes close to each other (both in the tree and geographically), then flattening preserves the geographic locality in the tree structure.

Our basic algorithm is designed to work with a single shared object on a binary tree for object retrieval protocols enforcing mutually exclusive access. We present extensions to this basic algorithm that allow nodes sharing multiple objects in $k$-ary trees through any protocol that runs on tree edges, to use our techniques for reduced access costs.

## 1.1 Related work and comparison to alternatives

The main goal of our work is to guarantee $O(\log n)$ access costs in a distributed tree structure. This issue has previously been addressed in the context of binary search trees and self-adjusting overlay networks.

### 1.1.1 Binary search trees

Our work is inspired by work on balancing binary search trees (BSTs) in a centralized system (e.g., [1, 29, 3, 11]), particularly the work of Sleator and Tarjan on *splay trees* [29]. Splay trees, detailed in the Appendix, are an elegant solution to efficient BSTs that achieve $O(\log n)$ amortized cost per search. When a node in the tree is accessed, splaying uses a "move to front" heuristic that brings the accessed item to the root of the tree, on the assumption that this item will be accessed again, while taking steps to balance the tree in the process.

It is not too difficult to design a distributed version of splaying to work in conjunction with an object retrieval protocol, particularly if splaying is done by the node holding the object: If the object retrieval protocol ensures mutually exclusive access to the object, splaying while holding the object ensures that concurrent

splaying will not take place, thereby substantially simplifying the design of the algorithm and precluding the need for expensive locking of parts of the tree. Thus the obvious adaptation of splaying to our setting is the following scheme: Upon retrieving the object, a node splays itself to the root of the tree, before releasing the object to the next node. This simple adaptation inherits the $O(\log n)$ amortized cost of the splay trees. So the question arises, why not simply implement and use splaying as a distributed algorithm?

One of the main concerns with employing splay trees in a distributed setting is the large number of messages that need to be exchanged. Even when the tree is perfectly balanced and the retrieving node is close to the current owner of the object, splaying would still rotate the retrieving node to the root, resulting in a high messaging cost for the retrieval.

Even in the centralized setting, this excessive reorganization of the tree has been a major concern and researchers have tried to counter this problem by developing variants of splaying that reduce the amount of restructuring performed for each access. *Semi-splaying* [29] uses less restructuring by bringing an accessed item only part way up to the root. However, as noted by the authors, semi-splaying adapts rather slowly to changing workloads. *Periodic splaying* [34] restructures the tree every fixed number of accesses, instead of restructuring with every access. Although this variant produces good results in the average case, it cannot handle the worst case well. *Randomized splaying* [2, 8] rotates an accessed node to the root only with a certain probability. A high probability results in too much restructuring and a low probability fails to preserve locality-based sub-structures in the tree: even when all the retrievals are within a small subtree, some of the nodes from this subtree may end up close to the root while others may not. There is an obvious trade-off between the number of messages sent in a series of restructuring operations and the effectiveness of these operations. We find a better point in this trade-off than other approaches have allowed.

Another concern with splaying is that although bottom-up splaying performs restructuring in rather local steps, the top-down splaying and top-down semi-splaying variants (see Appendix for different types of splay techniques) require distant nodes in the tree to form edges with each other. Such operations in a distributed system not only result in a high messaging cost but also make it virtually impossible for a routing service—i.e., a service that routes messages from one node to another node along tree edges, possibly hopping through many intermediate nodes—to adapt to these modifications quickly. Routing through the tree is an essential part of our target protocols, and any restructuring that does not allow the routing information to be updated in a distributed and scalable fashion is undesirable.

Unlike splaying, distributed versions of B-trees [3] and variants like the B-link trees [20] have been proposed and analyzed in literature [10, 18, 24]. However, this research has been done in the context of distributing large dictionaries across nodes on a network. Tree balancing is employed to distribute dictionary data uniformly across the participating nodes. This model is very different from ours because its goal is distributed load balancing and not efficient object sharing. The fundamental difference with our approach is that whereas distributed B-trees explicitly balance the tree whenever a data item is inserted or removed from the tree, flattening only performs "on-demand" balancing that dynamically restructures the tree according to the workload to maintain an $O(\log n)$ retrieval cost. Moreover, these distributed B-tree proposals do not support an object retrieval protocol that can be used for providing consistency of a modifiable shared object as required in our setting.

### 1.1.2 Overlay networks

In more recent years, much research has been done on self-adjusting overlay networks ([30, 25, 27, 16, 17, 22], to name a few) which retain an $O(\log n)$ diameter as nodes join and leave the overlay. The restructuring is either done during the join and leave operations or is done periodically [30, 25, 27], where the frequency may depend on observed parameters like failure rate [22]. Our focus in this paper is somewhat different:

given an arbitrary tree topology our goal is to guarantee $O(\log n)$ amortized retrieval costs. Therefore, we do not expect anything from the join or leave procedures other than that they keep the tree connected. To improve the retrieval costs, our tree adjusts itself according to the workload. So for example, if a certain subtree is never used by the application running on our overlay then that subtree is never restructured. This approach may result in more or less restructuring than other self-adjusting overlay proposals depending on the workload. However, once adjusted, our scheme achieves the optimal structure for this workload[1], whereas other self-adjusting overlays may not. Note that our algorithms achieve these properties for any workload while retaining the worst case amortized cost of $O(\log n)$.

### 1.2 Contributions

Our contributions in this paper are threefold: First, to our knowledge, our work is the first that attempts to restructure a distributed tree according to the workload to reduce the amortized cost of object retrievals within the tree. We believe such an algorithm could improve the worst case performance of a large group of distributed protocols.

Second, we present new bottom-up and top-down restructuring primitives that exploit the fact that our algorithm need not be order-preserving (unlike binary search trees). The resulting primitives also achieve $O(\log n)$ amortized costs like the splaying counterparts but are optimized for the distributed setting: *Top-down semi-flattening* has a much lower message complexity than the splaying variants, and both top-down and *bottom-up flattening* techniques allow nodes to make local restructuring decisions, permitting them to easily update their local routing tables to reflect the new tree topology.

Finally, we present a novel approach that combines bottom-up flattening and top-down semi-flattening schemes in one simple algorithm. This algorithm adapts the amount of restructuring to the workload and restructures the tree on an "on-demand" basis. The result is a restructuring scheme that has a low messaging cost but still adapts quickly to workloads. Our algorithm also preserves locality in the tree structure if the workload permits, a desirable property for tree topologies that are based on network metrics or geographical information [35, 9].

To see the effectiveness of our algorithms in a real-world setting, we implement the Arrow distributed directory protocol [7] and perform experiments with it on PlanetLab, using both a static tree and a tree implementing flattening algorithms. Results from these experiments show that the worst case performance of the Arrow protocol is improved by several orders of magnitude when used with flattening. We also compare our scheme against different splaying variants and show that flattening is much better optimized for the distributed setting.

## 2 System model

Our system consists of a set of $n$ nodes distributed across a network and initially structured as a rooted, binary, unordered tree. This tree is a logical overlay network with vertices representing nodes in the system and edges representing overlay edges. Each node $v \in V$ is initialized only with the identities of its neighbors in the tree, i.e., a parent pointer (initialized to the distinguished value "$\perp$" for 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

[1]The proof of this guarantee appears in [29] as the *static optimality theorem* which is a direct result of the *access lemma*. We prove the access lemma here (see Lemmas 1 and 2) for our algorithms and therefore the static optimality theorem also applies.

between them are reliable: nodes do not fail, and each RPC completes in a finite but unbounded time, i.e., communication is asynchronous.
Nodes access mobile objects that are used in application-specific protocols. For brevity, we deal with a single mobile object; extensions to multiple objects are discussed in Section 8. The node that initiates a *retrieval* of the object is called a *requestor*. We say a request *terminates*, when it reaches its *responder*, i.e., the node that will release the object to be sent to the requestor. The responder initiates the *transfer* (when it does not need the object anymore), which follows the unique path in the tree from the responder to the requestor. When the object arrives at the requestor, the retrieval is complete. There may be several concurrent requests but at most one transfer in the tree at a time. Object retrieval protocols enforcing mutually exclusive access satisfy these properties. All other mechanisms related to object retrievals are protocol specific. We make no assumptions about these details.
Our algorithm restructures the tree during the object transfer to avoid concurrent restructuring, as transfers do not overlap. Each transfer results in a new tree. We use $T_0$ to denote the initial tree before any retrievals begin.

# 3 Properties

We present a distributed algorithm that offers the following properties, while maintaining the nodes in a tree structure:

G1. The message complexity of $m$ retrieval operations is $O(m \log n)$.

G2. All messages exchanged in a retrieval are confined to the subtree containing the requestor and the responder, and the parent of that subtree. As a corollary, we achieve the following property: Let $T_i = (V_i, E_i)$ denote the tree after the $i^{th}$ retrieval has completed such that there are no outstanding retrievals in the tree and node $v_0 \in V_i$ owns the object. Let $v_1, ..., v_k$ be the $k$ nodes that start the $k$ subsequent retrievals whose completion results in the tree $T_{i+k}$. Let $T' = (V', E')$ be the smallest subtree of $T_i$ such that $v_0, ..., v_k \in V'$. Then the message complexity of each of these $k$ retrievals is proportional to (i.e., is a small constant multiple of) $|V'|$.

Our algorithm guarantees an $O(\log n)$ amortized cost per retrieval (G1). In addition the algorithm takes advantage of the locality in the workload, i.e., if the requestor and the responder are close to each other, then only a small amount of restructuring will be done because all restructuring will be confined to a small subtree (G2). This results in a small messaging cost for individual retrieval operations. Furthermore, this property allows us to preserve geographical mappings in the tree, if all requestors and responders belong to a certain geographical region.

# 4 Self-adjusting distributed trees

## 4.1 Key insight

Splay 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 start from any node in the tree. So a better heuristic is to move the responders close to the requestors. One way to achieve this would be to move both the requestors and responders close to the root, but this might be an overkill and would require excessive restructuring. Instead, we rotate the requestors and the responders close to the root of the smallest

subtree that contains both of them. This scheme has the advantage of minimizing the restructuring in the tree if the requestor and the responder are close to each other already, while still achieving the $O(\log n)$ amortized cost. We implement this scheme by restructuring along the transfer path from the responder to the requestor employing both *bottom-up flattening* and *top-down semi-flattening* techniques in one algorithm. 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.

Most existing restructuring techniques including splay trees 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 variants) flattening algorithms that use these primitives.

## 4.2 Bottom-up flattening

Our first algorithm is a bottom-up scheme that restructures the tree whenever the object moves up an edge during its transfer from the responder $t$ to the requestor $r$. Bottom-up flattening starts from $t$ and proceeds to the highest node in the transfer path to $r$. In case $t$ is the highest node, no restructuring is done. The result of bottom-up flattening is to bring $t$ to the root of the subtree that contains $r$.

### 4.2.1 Preferred rotation primitive

We define a variation of the rotation primitive for bottom-up flattening. For each rotation performed by the responder $t$ 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 preferred rotations until it rotates over the highest node in the path to $r$.

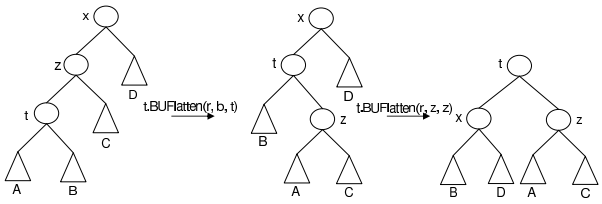


Figure 1: Bottom-up flattening: $t$ rotates keeping the preferred child. Preferred child is the one that $t$ last rotated over. For the first rotation any child may be preferred.

### 4.2.2 Bottom-up flattening algorithm

Figure 2 shows the distributed algorithm that implements bottom-up flattening. We denote the variables encoding global state at a node $y$ using the prefix "$y.$", e.g., $y.\mathsf{parent}$. Variable names without the prefix denote temporary state that is deleted once this invocation is over. We assume the existence of a routing service $R$ that can be queried by nodes to find the next node, and if a node itself is the highest node, in the path from a responder to a requestor. We describe a minimal distributed routing service that achieves these goals in Section 5.

```
1.  t.BUFlatten(r, b, w)                          /* r: requestor, b: preferred child, w: former child of t.parent */
2.      a ← t.children \ {b}                      /* a is the child not preferred */
3.      z ← t.parent                              /* z is the current parent */
4.      t.children ← {t.children \ {a}} ∪ {z}     /* replace child a with z */
5.      [gParent, isHigh] ← z.rotateEdge(t, r, w, a)  /* z replaces its child w with a and sets z.parent to t */
6.      t.parent ← gParent                        /* set new parent to old grand-parent */
7.      a.setParent(z)                            /* a.parent now points to z */
8.      if isHigh                                 /* 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: responder, r: requestor, w: child to replace, a: new child */
12.     x ← z.parent                              /* x is my current parent */
13.     z.parent ← t                              /* set t as new parent */
14.     z.children ← {z.children \ {w}} ∪ {a}     /* replace child w with a */
15.     return [x, R.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 ← {x.children \ {z}} ∪ {t}     /* replace child z with t and return */

18. a.setParent(z)                                /* z: new parent */
19.     a.parent ← z                              /* set parent to z and return */
```

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

The responder $t$ initiates bottom-up flattening by invoking $t$.BUFlatten$(r, b, t)$. 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 message (line 5) results in $z$ setting $z$.parent to $t$ and adding $t$'s non-preferred child $a$ to $z$.children replacing $t$. 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.

#### 4.2.3 Discussion

Each preferred rotation requires 4 messages—two messages for each of the two RPCs (lines 5 and 7), except for the last rotation that requires 6 messages due to line 9. The most efficient implementation of bottom-up splaying (see Appendix) also requires 8 messages for moving up two steps. Although the messaging costs are similar, bottom-up flattening has a much simpler algorithmic logic than bottom-up splaying. Bottom-up flattening is also slightly more efficient than bottom-up splaying in terms of the amortized cost (the constant in the "big-oh" notation is smaller).

### 4.3 Top-down semi-flattening

Our second algorithm is a top-down scheme that restructures the tree whenever the object moves down an edge during its transfer. Top-down flattening starts at the highest node in the path from the responder $t$ to the requestor $r$. Since top-down flattening is preceded by bottom-up, this highest node is, in fact, the responder $t$. We bring $r$ close to $t$ via a top-down semi-flattening mechanism that brings $r$ part way up to $t$, while performing less restructuring than full flattening.

#### 4.3.1 Child swap primitive

Top-down semi-flattening is performed by repeating the step shown in Figure 3. $y$, $x$ and $a$ are in path from $t$ to $r$. Node $y$ swaps one of its children (root of subtree $C$) 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 $a$. Flattening is started by $t$ and stops if $r$ is the current node or a child of the current node.

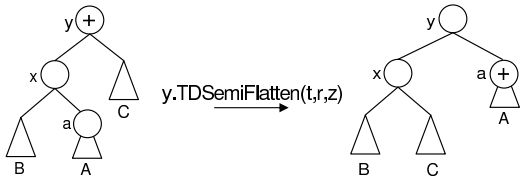


Figure 3: Top-down semi-flattening: $y$, $x$ and $a$ are in the path from $t$ to $r$. Next child swap is performed by $a$.

| | | |
|---|---|---|
| 1. | $y$.TDSemiFlatten$(t, r, z)$ | /* $t$: responder, $r$: requestor, $z$: my new parent */ |
| 2. | $y$.parent $\leftarrow z$ | |
| 3. | **if** $r \in \{y\} \cup y$.children | /* if I or my child is the requestor, then...*/ |
| 4. | **stop** | /* ...stop the restructuring */ |
| 5. | $x \leftarrow R$.nextNode$(y, t, r)$ | /* find the child that is in path from $t$ to $r$ */ |
| 6. | $c \leftarrow y$.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$(t, r, c)$ | /* $t$: responder, $r$: requestor, $c$: my parent's child not in path */ |
| 12. | $a \leftarrow R$.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.

#### 4.3.2 Top-down semi-flattening algorithm

Figure 4 shows the distributed algorithm for this scheme. The algorithm is initiated by $t$ as $t$.TDSemiFlatten$(t, r, t$.parent$)$. At each step, the current node $y$ and its child $x$ swap $y$'s child that is not

in the path between $t$ and $r$ with $x$'s child that is in this path (lines 9 and 13). The swapped children are notified of their new parents (lines 7 and 10).

#### 4.3.3 Discussion

Top-down semi-flattening approximately halves the depth of each node (relative to $t$) in the path from $t$ to $r$, after bottom-up flattening. As a result, semi-flattening brings $r$ closer to $t$. Compared to the splaying counter parts, top-down semi-flattening has a much lower messaging cost. In particular, each child swap requires 5 messages and moves two steps down the tree—two messages for each of the two RPCs in lines 7 and 8 and an additional message for the RPC in line 10 (this RPC may be non-blocking, and so we do not count its response against the latency of the child swap). In contrast, an optimal implementation of top-down semi-splaying (see Appendix) requires a number of messages ranging from 6 to 13 for each step. Furthermore, top-down semi-flattening performs only local pointer reassignments that allows simple updates for a routing service. Top-down semi-splaying makes it difficult for a distributed routing service to update the routes correctly after restructuring.

### 4.4 Hybrid flattening

#### 4.4.1 Hybrid flattening algorithm

Our main algorithm combines bottom-up flattening with top-down semi-flattening to restructure along the transfer path from the responder $t$ to the requestor $r$. 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 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).

| | | |
|---|---|---|
| 1. | $t$.HybridFlatten$(r)$ | /* $r$: requestor */ |
| 2. | **if** $R$.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 non-null then choose any */ |
| 7. | $t$.BUFlatten$(r,$ 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.

#### 4.4.2 Discussion

Hybrid flattening inherits its low message complexity and simplicity from the two constituent schemes. It achieves an amortized cost of $O(\log n)$ (G1), see Section 6 for a detailed analysis. The restructuring follows the transfer path from the responder to the requestor, achieving G2. Figure 6 shows an example tree where $r$ retrieves the object from $t$. Hybrid flattening brings $t$ and $r$ close to each other and in process, balances the remaining tree.

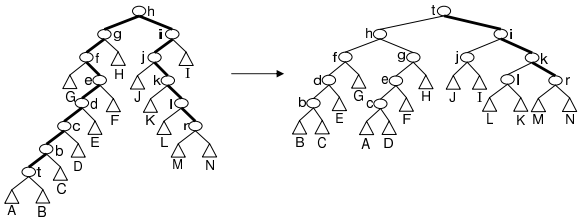


Figure 6: Hybrid flattening. Bold lines show the path between $t$ and $r$. Root of $A$ was the first preferred child.

# 5 Route management

In the flattening algorithms we assume a routing service $R$ that supports two types of queries: First, the $R.\mathsf{nextNode}(y, t, r)$ query is invoked by a node $y$ to obtain the node following $y$ in path from $t$ to $r$ in the current tree (lines 5 and 12 in Figure 4). Second, the $R.\mathsf{amHighNode}(y, t, r)$ query is invoked by a node $y$ to find out if $y$ is the highest node (i.e., closest to the root) in the path from $t$ to $r$ in the current tree (line 15 in Figure 2 and line 2 in Figure 5).

These two queries can be supported by a simple distributed routing service as follows: An instance of the routing service $R_y$ runs locally at every node $y$ and observes all requests and transfers passing through $y$. $R_y$ inserts an entry in the local routing table upon observing a request and uses this entry to answer $R_y.\mathsf{nextNode}$ and $R_y.\mathsf{amHighNode}$ queries during the corresponding transfer. The entry is deleted after the transfer passes through $y$. Each request contains the identity of the requestor $r$ and the identity of $y$'s neighbor $x$ that forwards this request to $y$. $y$ replaces $x$ with $y$ in the request and forwards this request towards its neighbor $z$ that is in direction of the responder ($z$ is identified using application specific mechanisms, e.g., see Section 7). Upon observing such a request, $R_y$ adds the following entry to the routing table: $\{r : z \rightarrow y \rightarrow x, b\}$ , i.e., the transfer destined for $r$ will come from $z$ and should be forwarded to $x$. $b$ is a boolean encoding whether $y$ is the highest node in the path or not. $R_y$ sets $b$ to $\mathsf{false}$ if either $z$ or $x$ is $y.\mathsf{parent}$ and to $\mathsf{true}$ otherwise.

This simple mechanism using backward pointers is enough to answer the two queries mentioned above. However, this routing information must be updated with restructuring to reflect the new tree. The rules for modifying the routing table with restructuring are simple since all restructuring is local and deterministic and transfers do not overlap. Some updates require neighbors to exchange their routing tables with each other, but not with any other nodes in the tree. Listing the whole set of rules is space consuming and uninteresting and is therefore avoided. As an example we list a subset of rules used by node $y$ when performing a child-swap step in top-down semi-flattening as shown in Figure 3, see Tables 1 and 2. $z$ is assumed to be $y.\mathsf{parent}$. Rules with an $(*)$ require $x$'s routing table; this can be piggy-backed on the return message sent in line 14 of Figure 4. As an example, the first rule in Table 1 states that if the transfer was coming from $z$ and going to $x$ and according to $x$'s routing table entry for this destination, $x$ was forwarding this to $a$, then $y$ can forward this to $a$ directly after the flattening ($a$ becomes a child of $y$ as a result of the restructuring) and $y$ is not the highest node in this path. $\perp$ implies that this entry is not required any more and can be deleted. Similar rules exist for all cases in our algorithms.

We note that all messages exchanged by neighboring routing services (to exchange routing tables) can be piggy-backed on existing messages as shown in the example above. Thus route management does not add to the message complexity of our algorithm. To compare with splay trees, if we were to use top-down semi-splaying instead of top-down semi-flattening, the distributed routing service would be much

more complicated than the one presented here and would require a much higher message complexity, as the restructuring in top-down semi-splaying is not localized (see Appendix for details on top-down semi-splaying).

Table 1: Modifications done by $y$ for routes to $x$

| Before | After |
|---|---|
| $(*)\ z \to y \to x \mid x \to a$ | $z \to y \to a, \mathsf{false}$ |
| $(*)\quad y \to x \mid x \to a$ | $y \to a, \mathsf{true}$ |
| $(*)\ c \to y \to x \mid x \to a$ | $x \to y \to a, \mathsf{true}$ |
| $(*)\ c \to y \to x \mid x \not\to a$ | $\perp$ |

Table 2: Modifications done by $y$ for routes to $c$

| Before | After |
|---|---|
| $z \to y \to c$ | $z \to y \to x, \mathsf{false}$ |
| $y \to c$ | $y \to x, \mathsf{true}$ |
| $(*)\ x \to y \to c \mid a \to x$ | $a \to y \to x, \mathsf{true}$ |
| $(*)\ x \to y \to c \mid a \not\to x$ | $\perp$ |

# 6 Amortized analysis

We use message complexity (number of messages exchanged) as the *cost* measure in our analysis. This is justified in a distributed setting as network latency is expected to dominate other factors like processing time. To allow comparison with splay trees, we use the potential method [32] for the amortized analysis as in [29]. 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:

$$\text{expense} = \text{actual cost} + \text{net increase in potential}$$

Using this definition, the total actual cost of a sequence of $m$ operations can be derived as:

$$\text{total actual cost} = \text{total expense} + \text{net decrease in potential} \tag{1}$$

Our proof strategy to bound the total actual cost 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 of $n$ nodes.

We begin by assigning a positive weight $w(x)$ to each node $x$ that remains fixed throughout the execution. Then define the size $\mathbf{s}(x)$ of a node $x$ to be the sum of weights of all nodes in the subtree rooted at $x$. We define the rank $\mathbf{r}(x)$ of $x$ as $\log(\mathbf{s}(x))$ (binary logarithms are used throughout). For each node $x$, we keep $\mathbf{r}(x)$ tokens on that node, thus the potential function is just the sum of the ranks of all nodes in the tree. As a measure of the actual cost, we charge 1 for each child swap and preferred rotation. We use $\mathbf{s}$ and $\mathbf{s}'$, $\mathbf{r}$ and $\mathbf{r}'$ to denote the sizes and ranks 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* $2(\mathbf{r}(t) - \mathbf{r}(r))$.

*Proof.* Top-down semi-flattening constitutes of child swaps. The expense of top-down semi-flattening is the sum of the expense costs of all the child swaps along the way. 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 $2(\mathbf{r}(y) - \mathbf{r}'(a))$. The sum of these child swap costs telescopes to $2(\mathbf{r}(t) - \mathbf{r}(r))$ if the path length between $t$ and $r$ is even and $2(\mathbf{r}(t) - \mathbf{r}(r'))$ if this length is odd; where $r'$ is the parent of $r$. The lemma holds in either case since $\mathbf{r}(r') \geq \mathbf{r}(r)$.
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 cost associated with a child swap is 1 so the expense is:

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

Now we need to prove the following:

$$1 + \mathbf{r}'(x) - \mathbf{r}(a) \leq 2(\mathbf{r}(y) - \mathbf{r}'(a))$$

or equivalently

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

This last inequality is true since $\mathbf{s}(y) \geq \mathbf{s}'(a) + \mathbf{s}'(x)$ and $\log a + \log b$ maximizes at -2 if $a + b \leq 1$ (due to the convexity of $\log$). □

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

*Proof.* Bottom-up flattening constitutes 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 constitutes of these pairs of preferred rotations, possibly followed by a single preferred rotation at the end if 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 amortized cost of a single preferred rotation is at most $2(\mathbf{r}'(t) - \mathbf{r}(t)) + 1$ and that of a pair of preferred rotations is at most $2(\mathbf{r}'(t) - \mathbf{r}(t))$. The sum of these costs telescopes and proves the lemma. We now prove our claim.
The actual cost of a single preferred rotation performed by $t$ over $z$ is 1. The expense is:

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

The actual cost of a pair of preferred rotations performed by $t$ over $z$ and then $x$ (see Figure 1) is 2. The amortized cost is:

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

Now we need to prove the following:

$$2 + \mathbf{r}'(z) + \mathbf{r}'(x) - 2\mathbf{r}(t) \leq 2(\mathbf{r}'(t) - \mathbf{r}(t))$$

or equivalently

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

This last inequality is true since $\mathbf{s}'(y) \geq \mathbf{s}'(z) + \mathbf{s}'(x)$ and $\log a + \log b$ maximizes at -2 if $a + b \leq 1$ (due to the convexity of $\log$). □

**Lemma 3.** *The net decrease in potential over any sequence of operations is at most* $\sum_{y=1}^{n} \log(\frac{W}{w(y)})$*, where* $W = \sum_{y=1}^{n} w(y)$.

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

**Theorem 1.** *The total actual cost of 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. 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 $\sum_{y=1}^{n} \log(\frac{W}{w(y)}) = n \log n$. Substituting these values in Equation 1 proves the result. □

**Theorem 2.** *The total actual cost of 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 $\sum_{y=1}^{n} \log(\frac{W}{w(y)}) = n \log n$. Substituting these values in Equation 1 proves the result. □

**Theorem 3.** *The total actual cost of 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. Since this 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 $\sum_{y=1}^{n} \log(\frac{W}{w(y)}) = n\log n$. Substituting these values in Equation 1 proves the result. □

# 7 Experiments

We have a complete implementation of the flattening algorithms and the distributed routing service described in previous sections. Our experiments use this implementation with the Arrow distributed directory protocol [7]. We modified the Arrow protocol so the transfers also follow the path through the tree; the Arrow protocol proposed in [7] uses the tree edges only to route requests and sends the transfers directly over the underlying network. This change was implemented using the routing service described in Section 5. The Arrow protocol maintains a forward pointer called an *arrow* on each node; this arrow points in the direction of the responder. These arrows also need to be maintained like the backward pointers described in Section 5 in the event of restructuring. This is also done through a set of rules similar to those used by the routing service. As with route management, the messages exchanged by neighboring nodes to keep the arrows consistent are also piggy-backed on the flattening protocol messages. We skip these details as they pertain specifically to the Arrow protocol.

We performed our experiments on PlanetLab [6]. Around 75 PlanetLab nodes located in North America were used for all experiments. These included Internet2, CAnet and university machines in the US and Canada. To control the sequence of requests (so we can construct worst cases and other distributions), we used one node external to the overlay tree as a "monitor". The monitor exchanged control messages with all nodes, e.g., to have nodes initiate a request or pull information about how long a retrieval operation took.

We performed two sets of experiments. For the first experiment, we constructed the worst case tree, i.e., a "line" with 75 nodes. Nodes farthest away from each other in the tree were made to alternate sending 25 requests each, so the object would "ping-pong" between these two nodes. We repeated this experiment on 10 different worst case trees, i.e., we always constructed a line but the position of nodes in this line was chosen randomly. Figure 7 plots the average time per retrieval against the number of retrievals performed for the "vanilla" Arrow protocol, i.e., when no restructuring is employed, and for the Arrow protocol with hybrid flattening. Each point is a mean of the results from 10 experiments. Our results show that hybrid flattening handles the worst case as claimed. The amortized cost of retrievals with hybrid flattening improves a great deal on the vanilla Arrow protocol even for a small sequence, e.g., 10 retrievals.

Our second experiment compares the performance of flattening with splaying in a tree structure that reflects network proximity and a workload that allows to take advantage of this property. In this experiment, we construct a line consisting only of nodes located at Berkeley (11 in total), we call this the "Berkeley tree". We then construct a "random" binary tree using the other PlanetLab nodes and join the Berkeley tree to one of the leaf nodes (see Figure 9). Initially, the object is located at the root of the Berkeley tree. A total of 50 requests are made by randomly chosen Berkeley nodes. Again, we perform this experiment 10 times, each time with a different random binary tree. Figure 8 plots the average time per retrieval against the number of retrievals. Each point in the plot is a mean of the 10 experiments. The different curves plot the performance of the Arrow protocol using different types of restructuring schemes. Splaying has a high initial cost since it rotates each Berkeley node to the root of the tree when this node completes its first retrieval (see Figure 10 for an example topology from one of our experiments, with each Berkeley node having completed

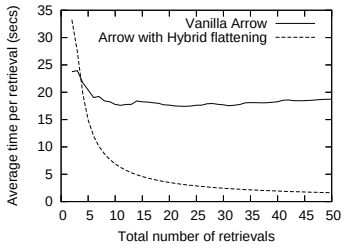


Figure 7: Amortized cost of the worst case.

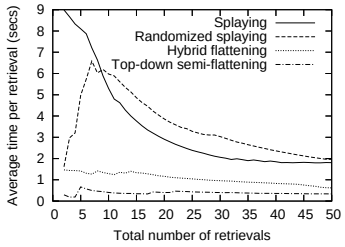


Figure 8: Amortized cost in a geographically mapped tree

at least one retrieval). Even when all Berkeley nodes have been rotated to the root, retrievals are somewhat expensive due to the large amount of restructuring required with every transfer. Randomized splaying does not have a high restructuring cost but moves only a subset of Berkeley nodes to the root resulting in some of the retrievals going through other nodes in the tree (see Figure 11). This situation gets better as more Berkeley nodes are rotated to the root. We used randomized splaying with a probability of 0.3 to splay after a retrieval. The choice is somewhat arbitrary, higher probability results in high restructuring costs and lower probability breaks the locality in the tree.

In contrast to the two splaying schemes, hybrid and top-down semi-flattening perform much better. We evaluate top-down semi-flattening individually as it can be used without bottom-up flattening in extensions to our protocol applicable in a more generic setting, see Section 8. In the flattening experiments, all restructuring is confined to the Berkeley nodes (see Figures 12 and Figure 13). For this particular workload top-down semi-flattening out-performs the hybrid case due to its low restructuring costs. In general, e.g., if the Berkeley tree had 1000 nodes, the semi-flattening scheme would not adapt as quickly to the workload as hybrid flattening. Hybrid flattening, however, will adapt to arbitrary workloads and has a performance comparable to the semi-flattening scheme for this particular workload too.

We illustrate an example topology used in the experiments documented in Figure 8. Initially, the tree looks as shown in Figure 9 where the black ovals show the nodes located at Berkeley. The subsequent topologies shown here are the result of using different types of restructuring, when each of the Berkeley nodes have completed at least one retrieval. Since Berkeley nodes make requests randomly, some nodes may have retrieved the object more than once. Figures 10, 11, 12 and 13 show this tree when using splaying, randomized splaying, hybrid flattening and top-down semi-flattening respectively.

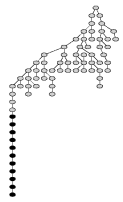

Figure 9: Initial topology. Black nodes are all at Berkeley

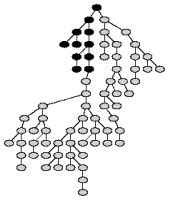

Figure 10: Splaying: after all Berkeley nodes requested at least once

# 8 Extensions and other applications

As presented, our algorithms work for a single object shared by nodes arranged in a binary tree running an object retrieval protocol that enforces mutually exclusive access. However, simple extensions to the basic algorithm presented earlier can result in variations that are much more generic and widely applicable to many different scenarios. Here we discuss these extensions and possible applications that can benefit from these variations.

## 8.1 Non-mutually exclusive protocols

Our algorithms avoid concurrent restructuring and in-efficient locking of the tree by restructuring only when the object is being transferred in a retrieval protocol that enforces mutual exclusion. One way to avoid locking large parts of the tree for protocols that do not enforce mutual exclusion, is to only use top-down semi-flattening. In this case, if two concurrent restructuring operations do not cross paths in the tree, then we do not have to deal with concurrency control. If they do cross paths in the tree, then the highest node (closest to root) that sees both of these operations can perform simple local concurrency control to avoid any inconsistencies in the tree structure. Our experiments with top-down semi-flattening (see Section 7) show that it performs well in a distributed setting, since it uses a very small number of additional messages for restructuring.

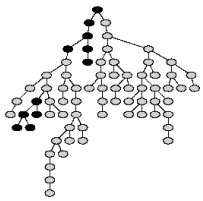

Figure 11: Randomized splaying: after all Berkeley nodes requested at least once

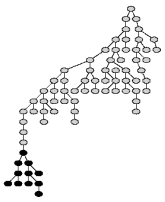

Figure 12: Hybrid flattening: after all Berkeley nodes requested at least once

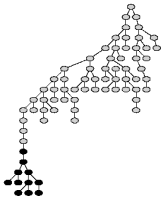

Figure 13: Top-down semi-flattening: after all Berkeley nodes requested at least once

### 8.2 Multiple objects

For simplicity, we described our algorithms assuming a single shared object. Multiple objects can easily be supported by maintaining a different logical tree for each shared object. This can result in some optimizations as well, e.g., only nodes interested in an object may join that object's tree. This allows nodes not interested in an object to avoid keeping any state or routing any messages for this object. It also allows the interested nodes to retrieve the object from each other without having to route through other nodes. This can greatly improve performance in applications where interest in the same object reflects network proximity.

### 8.3 K-ary trees

Our algorithms as described in the previous sections work only for a binary tree. However, extensions to $k$-ary trees are 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 transfer path—denote it as the *most significant node*. In the first step of Figure 1, root of subtree $A$ is the least significant node and $z$ is the most significant node whereas in Figure 3, root of subtree $C$ is the least significant node and $a$ is the most significant node. In case of a $k$-ary tree, the most significant node is still well-defined (the node in the transfer path) 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 which was in the transfer path of the most recent retrieval 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.

### 8.4 Applications

With these extensions, our self-adjusting tree can support many applications that run on overlays. We discuss two such application.

### 8.4.1 Application-level atomic multicast

Consider an application-level atomic multicast protocol where multiple sources may take turns to multicast, e.g., the source must hold a token before it is allowed to multicast and the token is shared among the sources using a retrieval protocol enforcing mutual exclusion (similar semantics have been achieved in [19, 4]). Our self-adjusting tree can form the overlay consisting of all the sources and receivers. As a source multicasts a batch of messages, top-down semi-flattening can be applied to balance the tree with this source as the reference root (nodes in the tree use the node where the messages are coming from as the parent and other neighbors as children). When a different source gets the token and starts to multicast its messages, the tree self adjusts and optimizes for the new source.

### 8.4.2 Content discovery

Without the restriction of mutually exclusive access, our algorithms can be applied to any object discovery and retrieval protocol in either the request or transfer phase. Our self-adjusting tree provides a generic substrate and any resource discovery protocols that use a tree structure can be laid on top of this substrate. If these protocols involve communication along the path from a node discovering the resource to the owner of the resource, then the top-down semi-flattening algorithm can be used. The effect will be to bring the source of the request closer to the owner of the resource while balancing the tree along the path. This can result in improved performance specially in applications where resources are mapped to nodes based on the semantics of the resource contents [31]. In these cases the same node can be expected to send future resource discovery requests (for semantically related resources) targeted to either the same resource owner or other nodes in its proximity.

## 9 Conclusions and future work

In this paper, we present a novel distributed algorithm that guarantees a worst case amortized message complexity of $O(\log n)$ for object retrievals in a distributed tree. In addition, our algorithm adjusts only portions of the tree in which retrievals occur; this is advantageous when the tree structure reflects network proximity. The existing algorithm works on binary trees. We believe it can be extended to $k$-ary trees using techniques similar to [28, 21]. We expect to investigate that in the future.

## References


[1] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Soviet Math. Dokl. 3*, pages 1259–1263, 1962.

[2] S. Albers and M. Karpinski. Randomized splay trees: theoretical and experimental results. *Information Processing Letters*, 81(4):213–221, 2002.

[3] R. Bayer. Symmetric binary b-trees: data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[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] 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.

[7] M. J. Demmer and M. P. Herlihy. The arrow distributed directory protocol. In *Proc. 12th International Symposium of Distributed Computing*, pages 119–133, 1998.

[8] M. Furer. Randomized splay trees. In *Proc. of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[9] L. Garces-Erice, E. W. Biersack, and P. A. Felber. Multi+: Building topology-aware overlay multicast trees. In *Proc. 5th International Workshop on Quality of Future Internet Services*, 2004.

[10] K. Gilon and D. Peleg. Compact deterministic distributed dictionaries. In *Proc. of the Annual ACM Symposium on Principles of Distributed Computing*, 1991.

[11] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, 1978.

[12] 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.

[13] M. Herlihy. The aleph toolkit: Support for scalable distributed shared objects. In *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, 1999.

[14] M. Herlihy and S. Tirthapura. Self stabilizing distributed queuing. *Lecture Notes in Computer Science*, 2180:209–223, 2001.

[15] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Ordered multicast and distributed swap. *Operating Systems Review*, 35(1):85–96, 2001.

[16] S. Jain, R. Mahajan, D. Wetherall, and G. Borriello. Scalable self-organizing overlays. Technical Report UW-CSE 02-02-02, University of Washington, 2002.

[17] 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 *Proc. of USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[18] 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.

[19] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, 1989.

[20] 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.

[21] C. Martel. Self-adjusting multi-way search trees. *Information Processing Letters*, 38(3):135–141, 1991.

[22] R. M. Miguel. Controlling the cost of reliability in peer-to-peer overlays, 2003.

[23] 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.

[24] D. Peleg. Distributed data structures: a complexity oriented view. *Proc. 4th International Workshop on Distributed Algorithms*, pages 71–89, 1990.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM Conference*, 2001.

[26] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, Feb. 1989.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware*, 2001.

[28] M. Sherk. Self-adjusting k-ary search trees. *Journal of Algorithms*, 19(1):25–44, 1995.

[29] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

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

[31] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. Conference on Applications, technologies, architectures, and protocols for computer communications*, 2003.

[32] R. E. Tarjan. Amortized computational complexity. *SIAM J. Appl. Discrete Math*, 6:306–318, 1985.

[33] 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.

[34] H. E. Williams, J. Zobel, and S. Heinz. Self-adjusting trees in practice for large text collections. *SoftwarePractice and Experience*, 31(10):925–939, 2001.

[35] L. Xiao, A. Patil, Y. Liu, L. Ni, and A.-H. Esfahanian. Prioritized overlay multicast in ad-hoc environments. *IEEE Computer*, pages 67–74, 2004.

# Splay trees

Splay trees [29] are binary search trees that guarantee an amortized cost of $O(\log n)$ per access and therefore are as efficient as any form of uniformly balanced trees for sufficiently long sequence of accesses. Furthermore, they are as efficient as static optimum search trees for sufficiently long sequence of accesses. Splay trees use heuristics to restructure the tree with each access operation. This restructuring (called splaying) brings the accessed item to the root of the tree. Splaying can either be done starting from the accessed node and moving up to the root (bottom-up splaying) or starting from the root and moving down to the accessed node (top-down splaying):

## Bottom-up splaying

Bottom-up splaying consists of a number of zig-zig (see Figure 14) and zig-zag (see Figure 15) steps followed by a single zig (see Figure 16) step; symmetric cases are omitted.

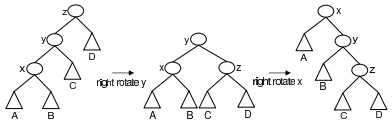


Figure 14: The zig-zig case in bottom-up splaying. Triangles denote subtrees.

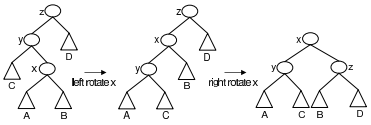


Figure 15: The zig-zag case in bottom-up splaying.


Figure 16: The zig case in bottom-up splaying.

## Top-down splaying

Top-down splaying maintains three different subtrees, a left subtree denoted $L$, a right subtree denoted $R$ and a middle subtree. The zig and zig-zig steps are shown in Figures 17 and 18, respectively. The zig-zag case (not shown) is just treated as two separate zig cases. The final step involves reassembling the tree from the three subtrees as shown in Figure 19. Top-down semi-splaying is a slightly modified version of top-down splaying that brings the accessed node only part way up to the root. The zig-zag and zig steps in top-down semi-splaying are the same as in top-down splaying. The zig-zig step uses the top tree $T$ and a special node called "Top" as shown in Figure 20.

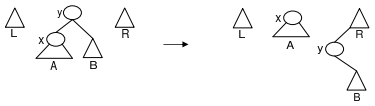


Figure 17: The zig case in top-down splaying. $y$ and $x$ are in path from root to the accessed node.

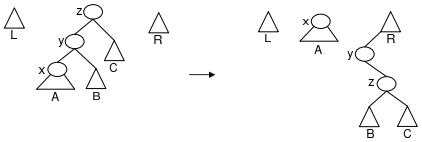


Figure 18: The zig-zig case in top-down splaying. A rotation is performed before $y$ is attached to $R$.

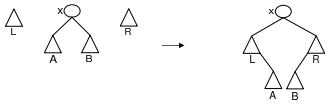


Figure 19: The reassembly step in top-down splaying.

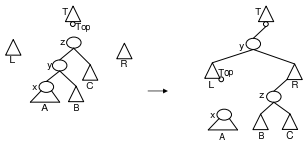


Figure 20: The zig-zig case in top-down semi-splaying.