The Arrow Distributed Directory Protocol

Michael J. Demmer[1] and Maurice P. Herlihy[2*]

[1] Tera Computer Company, Seattle WA 98102, miked@tera.com
[2] Computer Science Department, Brown University, Providence, RI 02912
herlihy@cs.brown.edu

Abstract. Most practical techniques for locating remote objects in a distributed system suffer from problems of scalability and locality of reference. We have devised the *Arrow distributed directory protocol*, a scalable and local mechanism for ensuring mutually exclusive access to mobile objects. This directory has communication complexity optimal within a factor of $(1 + \mathit{MST\text{-}stretch}(G))/2$, where $\mathit{MST\text{-}stretch}(G)$ is the "minimum spanning tree stretch" of the underlying network.

1 Introduction

Many distributed systems support some concept of *mobile objects*. A mobile object could be a file, a process, or any other data structure. For an object to be mobile, we require only that it can be transmitted over a network from one node to another. A mobile object "lives" on only one node at a time, and it moves from one node to another in response to explicit requests by *client* nodes. A *directory* service allows nodes to keep track of mobile objects. A directory must provide the ability to locate a mobile object (*navigation*), as well as the ability to ensure mutual exclusion in the presence of concurrent requests (*synchronization*).

This paper describes the *arrow distributed directory protocol*, a novel directory protocol being implemented as part of the Aleph toolkit [10], an distributed shared object system currently under development at Brown. The arrow directory protocol is designed to avoid scalability problems inherent in many directory services currently used in distributed shared memory systems. In this paper, we focus on proving the correctness of the protocol, and on analyzing its communication complexity. The service's data structures and protocols are extremely simple, and yet its communication complexity compares well to more complicated asymptotic schemes in the literature.

2 Motivation

Perhaps the simplest way to implement a directory service is to have each node broadcast each access request, and await the response from the node currently holding the object. Indeed, this approach is common in bus-based multiprocessor systems (e.g., [9]), where broadcasting can be accomplished efficiently. In

* Supported by AFOSR agreement F30602-96-0228 DARPA OD885.

distributed systems, however, broadcast is impractical, so a directory structure must provide a way to navigate and synchronize by point-to-point communication. It is equally impractical to have each node store each object's current location, since all nodes must be notified when an object moves. Additionally, since requests occur simultaneously, maintaining consistency in such schemes becomes difficult.

The most common directory protocol used in existing distributed shared memory systems is a *home-based* structure. Each mobile object is associated with a fixed node, termed that object's "home". The home keeps track of the object's location and status (e.g., busy or idle). When a client node requests an object, it sends a message to the object's home. The home sends a message to the client currently holding the object, and that client forwards the object to the requesting client node. The home can also enforce the necessary synchronization, by queuing concurrent requests. Home-based schemes are simple and easy to implement, and they have been observed to work well for small-to-medium scale systems. Nevertheless, such schemes suffer from problems of scalability and locality. As the number of nodes grows, or if an object is a "hot spot", that object's home is likely to become a synchronization bottleneck, since it must mediate all access to that object. Moreover, if a client is far from an object's home, then it must incur the cost of communicating with the home, even if the node currently holding the object is nearby.

One way to alleviate these problems is to allow an object's home to move. For example, Li and Hudak [15] proposed a protocol in which each object is initially associated with a particular node, but as an object is moved around, it leaves a virtual trail of *forwarding pointers*, starting from the original home. A limitation of this approach is that many requests for an object may still go through the original home, or end up chasing an arbitrarily long sequence of pointers. Additionally, if the object is close but the home is far, the client may still have to incur the large communication costs.

Our approach is also based on the idea of a trail of pointers, although we use them in a different way. Objects have no fixed homes, and synchronization and navigation are integrated into a single simple protocol. When a client requests an object from another client, all messages are sent through direct or nearly-direct paths, preserving locality, and permitting us to give explicit worst-case bounds on the protocol's communication complexity.

3 The Directory Structure

In this section, we give an informal definition of the arrow directory protocol, together with examples illustrating the interesting aspects of its behavior. A more formal treatment appears in Section 4.

For brevity, we consider a directory that tracks the location of a single object. We model a distributed system in the usual way, as a connected graph $G = (V, E)$, where $|V| = n$. Each vertex models a node, and each edge a two-way reliable communication link. A node can send messages directly to its neighbors, and indirectly to non-neighbors along a path. Each edge is weighted with a communication *cost*. The cost of sending a message from one node to another along a particular path is just the sum of that path's edge costs. The *distance* $d_G(x, y)$ is the cost of the shortest path from x to y in G. The network is *asynchronous* (steps are interleaved in an arbitrary order) but *reliable* (every node eventually takes a step and every message is eventually delivered). We assume the network provides a *routing service* [7, 21] that allows node v to send a message to node u with cost $d_G(u, v)$.

The *arrow directory* is given by a minimum spanning tree T for G. Each node v stores a directory entry $link(v)$, which is either a neighbor of v in T, or v itself. The meaning of the link is essentially the following: if $link(v) = v$, then the object either resides at v, or will soon reside at v. Otherwise, the object currently resides in the component of the spanning tree containing $link(v)$. Informally, except for the node that currently holds an object, a node knows only in which "direction" that object lies. If T has maximum degree Δ, then the directory requires $n{\cdot}\log(\Delta)$ bits of memory to track the object (some techniques for memory reduction are discussed below).

The entire directory protocol can be described in a single paragraph. The directory tree is initialized so that following the links from any node leads to the node where the object resides. When a node v wants to acquire exclusive access to the object, it sends a $find(v)$ message to $u_1 = link(v)$ and sets $link(v)$ to v. When node u_i receives a $find(v)$ message from node u_{i-1}, where $u_{i+1} = link(u_i)$, it immediately "flips" $link(u_i)$ to u_{i-1}. If $u_{i+1} \neq u_i$, then u_i forwards the message to u_{i+1}. Otherwise, u_i buffers the request until it is ready to release the object to v. Node u_i releases the object by sending a $move(v)$ message containing the object directly to v, without further interaction with the directory.

Despite the protocol's simplicity, it has a number of non-trivial properties that we believe to be of both practical and theoretical interest. Before analyzing this behavior, it is helpful to review some simple examples. Figure 1 shows a properly initialized directory, where directed edges correspond to the directory links. The object resides at node u (indicated by a square). When node v requests the object, it sends a $find(v)$ message to $u_1 = link(v)$, and then sets $link(v)$ to v. When u_1 receives the message from v, it forwards it to $u_2 = link(u_1)$, and sets $link(u_1)$ to v. Continuing in this way, each vertex thus "flips" the link to point in the direction from which the $find(v)$ message arrived. Figure 2 illustrates the directory state after three steps. Links changed by the protocol are shown in gray: v's link points to itself, u_1 points to v and u_2 points to u_1.

As shown in Figure 3, the $find(v)$ message continues to flip links until it arrives at u, where the object resides (marked by $link(u) = u$). When u receives the $find(v)$ message, it responds by sending the object in a $move((v)$ message directly to v. This response takes the shortest path, and does not affect the directory structure. As illustrated in Figure 4, after v has received the object, all links in the graph again lead to the object's new location, so any subsequent *find* messages will be directed there. The cost of acquiring access to the object is

$$d_T(u, v) + d_G(v, u).$$

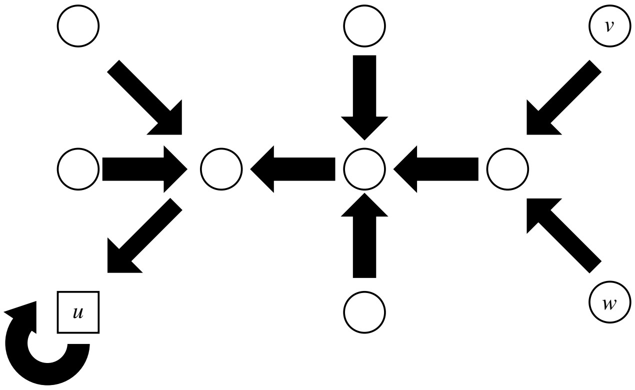

Fig. 1. Initial Directory State

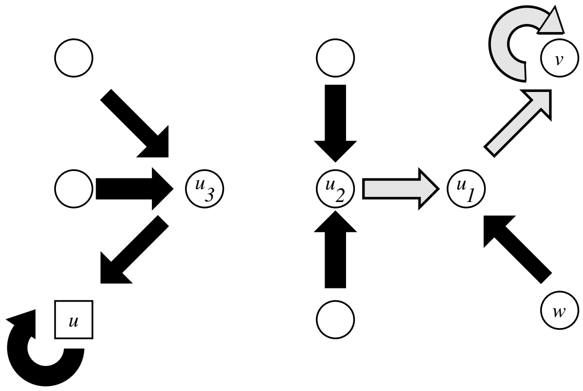

Fig. 2. Threee steps after u issues *find*(u) message

The term $d_T(u, v)$ is the cost of sending the *find*(v) message from u to v through the directory tree, and the term $d_G(v, u)$ is the cost of sending the *move*(v) message directly through the network. Notice the locality of this interaction: the message traffic affects only the nodes between u and v in the directory tree and network.

We now turn our attention to a concurrent example in which multiple nodes try to acquire the object at the same time. As described above, as a *find* message traverses the directory tree, it flips each link to point back to the neighbor from which the *find* originated. If two *find* messages are issued at about the same time, one will eventually cross the other's path, and be "diverted" away from the object and toward its competitor. Figure 2 illustrates this behavior. If any of the nodes on the right half of the tree requests the object, then its *find* message will follow the links to v, whereas nodes on the left half of the tree will follow the links to u.

For example, suppose w requests the object while v's *find* is still in progress.

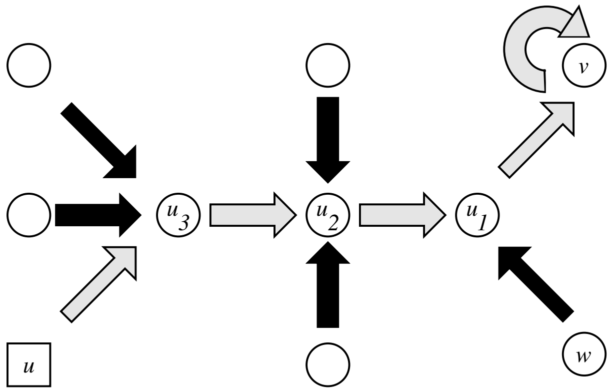

Fig. 3. Find message received

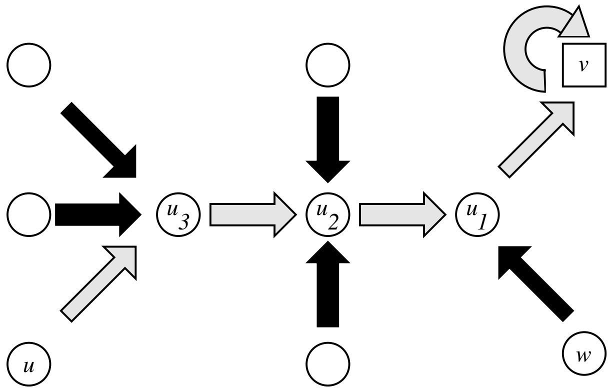

Fig. 4. Move message received

This second *find* will be diverted to v. When v receives w's message, it will not respond until it is has received the object and is ready to release it. This directory state is illustrated in Figure 5, where w's request is blocked at v.

Now suppose that node z issues a *find* which arrives at u, where it is buffered (u is not finished with the object). The *find* message from v is then diverted to z, as illustrated in Figure 6. This example illustrates how the arrow directory integrates synchronization and navigation in a natural way. In a quiescent state (when no messages are in transit), the directory imposes a distributed queue structure on blocked *find* requests. When z completes its *find*, after flipping its links, it blocks at u. Similarly, v blocks at z, and w at v. These blocked requests create a distributed queue where each *find* is buffered at its predecessor's node. When u releases the object, it goes directly to z, then v, then w. This distributed queue structure is valuable from a practical perspective, for several reasons. First, it ensures that no single node becomes a synchronization bottleneck. Second, if there is high contention for an object, then each time that

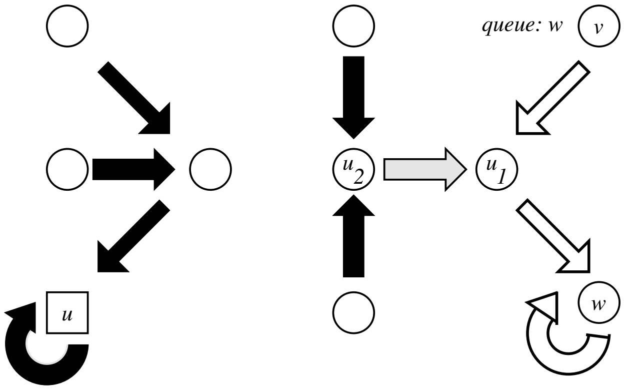

Fig. 5. w's request blocked at v

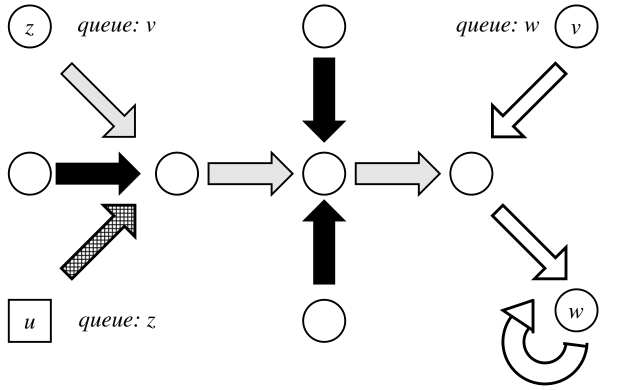

Fig. 6. Deflected find

object is released, that node will already have a buffered *find* request. In the limit, the cost of delivering *find* messages is hidden by the local computation times, and the protocol's performance approaches optimal (repeated local computation followed by direct object delivery). Third, the queue structure ensures locality: each *find* message takes a direct path through the spanning tree to its predecessor's node, with no detour to a home node.

The communication complexity of the concurrent execution in which the object goes from u to z to v to w is

$$(d_T(z,u) + d_G(u,z)) + (d_T(v,z) + d_G(z,v)) + (d_T(w,v) + d_G(v,w)).$$

The first expression in each parenthesized term is the cost of sending the *find* message from each node to its predecessor via the directory tree, and the second is the cost of sending the *move* message directly from each node to its successor. There are two important points to notice about this communication cost. First, there is no synchronization overhead: the communication cost is the same as a serial execution in which each node issues its *find* only after its predecessor is ready to release the object. Second, the inherent cost of the transfer from u to z is $d_G(z, u) + d_G(u, z)$, the inherent cost of delivering matching *find* and *move* messages. The arrow directory replaces the first term with $d_T(z, u)$, suggesting that the navigation cost of the arrow directory approaches optimality to the degree that $d_T(z, u)$ approaches $d_G(z, u)$. We elaborate on these observations in the next section.

4 Analysis

In this section, we describe the algorithm using a simplified version of the I/O automaton formalism of Lynch and Tuttle [17], we sketch safety and liveness issues, and evaluate the algorithm's cost.

4.1 Description

Recall that the network is a connected graph $G = (V, E)$, where $|V| = n$. There are two kinds of messages: node v issues a $\mathit{find}(v)$ message to request access to an object, and $\mathit{move}(u)$ to transfer ownership to u. Each node v has following attributes: $\mathit{link}(v)$ is a node, $\mathit{queue}(v)$ is either a node or $\perp$, and $\mathit{owner}(v)$ is a boolean. Each pair of nodes (u, v) has a an associated $\mathit{pending}(u, v)$, which is a set of messages that have been sent but not delivered (i.e., in transit, or queued at the sender or receiver).

A node v or link $\mathit{link}(v)$ is *terminal* if $\mathit{link}(v) = v$. The directory is initialized so that there is exactly one node v_0 such that v_0 is terminal and $\mathit{owner}(v_0)$ is true, the remaining non-terminal *link* pointers form an oriented tree, and all $\mathit{pending}(u, v)$ sets are empty. Message delivery is reliable, but not necessarily FIFO. A node initiates a *find* operation by sending a *find* message to itself For brevity, we assume that a node w has only one $\mathit{find}(w)$ message in the network at a time.

We use T to denote the undirected tree, and L to denote the directed graph induced by non-terminal links. The algorithm has the property that T does not change, but L does. The algorithm is defined by a number of transitions. The first part of each transition is a precondition that must be satisfied for the transition to occur, and the second part describes the state of the directory after the transition. A primed component indicates how a component's state changes after the transition.

Several transitions require a node atomically to remove a message from a network link, undergo an internal transition, and insert the message in another link. In practice, this atomicity is realized simply by requiring each node to finish processing one message before it starts processing the next.

The first transition says that when $\mathit{find}(w)$ is sent from node u to a non-terminal node v, v flips its *link* pointer to u and forwards the message to its old link.

pre: $\quad \mathit{find}(w) \in \mathit{pending}(u, v)$
$\quad\quad\quad \mathit{link}(v) \neq v$
post: $\quad \mathit{pending}'(u, v) = \mathit{pending}(u, v) - \{\mathit{find}(w)\}$
$\quad\quad\quad \mathit{pending}'(v, \mathit{link}(v)) = \mathit{pending}(v, \mathit{link}(v)) \cup \{\mathit{find}(w)\}$
$\quad\quad\quad \mathit{link}'(v) = u$

If, however, v is a terminal node, then w is enqueued behind v.

pre: $\quad \mathit{find}(w) \in \mathit{pending}(u, v)$
$\quad\quad\quad \mathit{link}(v) = v$
post: $\quad \mathit{pending}'(u, v) = \mathit{pending}(u, v) - \{\mathit{find}(w)\}$
$\quad\quad\quad \mathit{link}'(v) = u$
$\quad\quad\quad \mathit{queue}'(v) = w$

We do not explicitly model whether the object is actively in use. Instead, we treat the owner's receipt of a *find* and its corresponding *move* as distinct events, separated by an arbitrary but finite delay. If the current owner's queue is non-empty, then it may relinquish ownership and move the object directly to the waiting node.

pre: $\quad \mathit{owner}(v) = \mathit{true}$
$\quad\quad\quad \mathit{queue}(v) \neq \perp$
post: $\quad \mathit{pending}'(v, \mathit{queue}(v)) = \mathit{pending}(v, \mathit{queue}(v)) \cup \{\mathit{move}(\mathit{queue}(v))\}$
$\quad\quad\quad \mathit{owner}'(v) = \mathit{false}$
$\quad\quad\quad \mathit{queue}'(v) = \perp$

When a waiting node receives the object, that node becomes the new owner.

pre: $\quad \mathit{move}(v) \in \mathit{pending}(u, v)$
post: $\quad \mathit{owner}'(v) = \mathit{true}$
$\quad\quad\quad \mathit{pending}'(u, v) = \mathit{pending}(u, v) - \{\mathit{move}(v)\}$

4.2 Safety

Our proof of the directory's safety properties is an invariance argument. At all times, we show that there exists a well-defined *path* in L from any node that leads either to the object's current owner, or to another node that has requested the object. Because *find* messages follow these paths, any message that reaches the end of a path will either acquire ownership of the object, or join the queue waiting for the object.

First, we need to check that such paths exist.

Lemma 1. *The following property is invariant:*

$$\mathit{find}(w) \in \mathit{pending}(u, v) \Rightarrow \mathit{link}(u) \neq v \wedge \mathit{link}(v) \neq u. \tag{1}$$

Proof. Initially, the property holds vacuously, because no messages are in transit. Consider the first transition to violate the property. There are two cases to consider. In the first case, suppose the directory enters a state where

$$\mathit{find}(w) \in \mathit{pending}'(u, v) \wedge \mathit{link}'(u) = v.$$

Immediately before this transition, we claim that

$$\mathit{find}(w) \in \mathit{pending}(v, u) \wedge \mathit{link}(u) = v,$$

so the property was already violated. First, $\mathit{find}(w)$ is in $\mathit{pending}(v, u)$, because otherwise $\mathit{link}'(u)$ would not be set to v. Second, $\mathit{link}(u)$ must be v, because otherwise the message would not be forwarded to v.

In the second case, suppose the directory enters a state where

$$\mathit{find}(w) \in \mathit{pending}'(u, v) \wedge \mathit{link}'(v) = u.$$

Immediately before this transition, we claim that

$$\mathit{find}(w) \in \mathit{pending}(v, u) \wedge \mathit{link}(v) = u,$$

so the property was already violated. First, $\mathit{find}(w)$ is in $\mathit{pending}(v, u)$, because otherwise $\mathit{link}'(u)$ would not be set to v. Second, $\mathit{link}(v)$ must be u, because the transition does not change $\mathit{link}(v)$.

Lemma 2. *The directed graph L induced by the non-terminal links is always acyclic.*

Proof. Because T is a tree, it suffices to show the directory cannot enter a state where $\mathit{link}(u) = v$ and $\mathit{link}(v) = u$ for distinct nodes u and v. Consider the first transition that sets $\mathit{link}'(v) = u$ while $\mathit{link}(u) = v$. Any such transition is enabled only if $\mathit{find}(w) \in \mathit{pending}(u, v)$ and $\mathit{link}(u) = v$, which is impossible by Lemma 1.

Definition 3. The *path* from a node is the directed path in L from that node to a terminal node.

Definition 4. The *target* of v, denoted $\mathit{target}(v)$, is the terminal node at the end of its path. If $\mathit{find}(w)$ is a message in $\mathit{pending}(u, v)$, then that message's target is $\mathit{target}(v)$.

Lemma 2 guarantees that these notions are well-defined. A node is a *waiter* if it has requested an object, but has not yet become the owner of that object.

We are now ready to prove our principal safety result:

Theorem 5. *The path from any node always leads either to the current owner or a waiter.*

Proof. Any node v partitions the set of nodes G into A_v and B_v, where A_v consists of nodes whose paths include v, and B_v consists of the rest. For every u in A_v, $\mathit{target}(u) = \mathit{target}(v)$.

Consider the directory state immediately before a transition "flipping" $\mathit{link}(v)$ to u. By Lemma 2, $u \in B_v$. After the transition, for every w in B_v, $\mathit{target}'(w) = \mathit{target}(w)$, so $\mathit{target}(w)$ remains an owner or a waiter. For every z in A_v (including v itself), $\mathit{target}'(z) = \mathit{target}'(u) = \mathit{target}(u)$, so $\mathit{target}(z)$ remains an owner or a waiter.

4.3 Liveness

Liveness is also based on invariance arguments. Liveness requires that transitions occur *fairly*: any transition whose precondition remains true will eventually occur. (For example, any message in $pending(u, v)$ will eventually be removed.) We will show that when a *find* message is in transit, that message "traps" its target in a component L_1 of the link graph L: other *find* and *move* messages can move the target within L_1, but the target cannot leave L_1. Each transition that advances the *find* message shrinks L_1 by at least one node, so after at most n advances, L_1 consists of a single node, and the *find* message must reach its target.

Theorem 6. *Each find(w) message will be delivered to its target in n steps or fewer.*

Proof. Suppose $find(w)$ is in $pending(u, v)$. Deleting the edge (u, v) from the undirected tree T splits T into two disjoint trees, $before(w)$ (containing u) and $after(w)$ (containing v). Lemma 1 states that as long as $find(w)$ is in $pending(u, v)$, then $link(v) \neq u$ and $link(u) \neq v$, so the message's target will remain in $after(w)$. (Other *find* requests may still move the target within $after(w)$.)

If a transition delivers $find(w)$ to its target, we are done. A transition that moves the message from $pending(u, v)$ to $pending'(v, link(v))$ induces a new partition $before'(w)$ and $after'(w)$, where $v \notin after'(w)$. Because each $after'(w)$ is strictly smaller than $after(w)$, the $find(w)$ message will arrive at its target after at most n steps.

4.4 Complexity

How should we evaluate the communication cost of this protocol? Following Peleg [20], it is natural to compare our communication complexity to that of an *optimal* directory for which synchronization and navigation are free. The *optimal* directory accepts only serial schedules (so it pays nothing for synchronization), and delivers each *find* and *move* message directly (so it pays nothing for navigation).

An *execution* is a sequence of atomic node steps, message sends, and message receipts. An execution is *complete* if every *find* message has a matching *move*. and an execution is *serial* if the receipt of the i-th *move* message precedes the sending of each $(i + 1)$-st *find*. The next lemma states that we can restrict our attention to complete serial executions.

Lemma 7. *If E is any complete concurrent execution of the arrow directory protocol, then there exists a complete serial execution E' that serializes requests in the same order, sends the same messages, and leaves the directory in the same state.*

Define the *MST-stretch* of G, denoted $MST\text{-}stretch(G)$, to be

$$\max_{u,v \in V} \frac{d_T(u, v)}{d_G(u, v)}.$$

This quantity measures how far from optimal a path through the minimum spanning tree can be. The MST-stretch can be n in the worst case (a ring), but is likely to be much smaller in realistic networks (such as LANs, or the Internet).

Consider a serial execution in which $v_0, \ldots, v_\ell$ successively acquire the object. In the arrow directory, each *find* message traverses the spanning tree T, with cost $d_T(v_i, v_{i+1})$, while in the optimal directory, the message goes directly from v_i to v_{i+1}, with cost $d_G(v_i, v_{i+1})$. The ratio of these quantities is no more than $\mathit{MST\text{-}stretch}(G)$. In both the arrow and optimal directories, the *move* message incurs cost $d_G(v_i, v_{i+1})$, so this ratio is 1. (From a practical perspective, it is worth emphasizing that we are not hiding any constants: these bounds really are $\mathit{MST\text{-}stretch}(G)$ and 1, not $O(\mathit{MST\text{-}stretch}(G))$ or $O(1)$.)

As a result, for any complete serial execution, the ratio of communication costs for the arrow directory and the optimal directory is bounded by

$$\frac{1 + \mathit{MST\text{-}stretch}(G)}{2}.$$

The lower the network's MST-stretch, the closer the communication cost of the arrow directory is to optimal.

5 Discussion

Our discussion so far has focused on the safety, liveness, and communication cost of an idealized directory scheme. We now briefly address other issues of practical interest.

One sensible way to reduce the directory's memory consumption and message traffic is to organize the directory as a two-level structure. The network is partitioned into neighborhoods of small diameter, where the arrow directory is maintained at a per-neighborhood granularity, and a home-based directory is used within each individual neighborhood. (For example, the neighborhood could be a local subnetwork, and the node its gateway.) The resulting hybrid scheme trades communication cost against memory consumption and directory tree traffic. As discussed below, Peleg [20] and Awerbuch and Peleg [3] have also proposed multi-level directory structures.

Note that only *find* messages go through the directory tree — the network's routing service handles all other traffic, such as *move* messages, remote procedure calls, etc. *Find* messages for different objects that arrive together can be combined before forwarding.

We have assumed for simplicity that every node is initialized with a directory entry for each object. It is more practical to initialize an object's directory links in the following "lazy" manner. Before a node can request an object, it must first acquire that object's name. When a node u receives a message from v containing the name of an unfamiliar object, it sends a message through the directory tree to v initializing the intermediate missing links. This lazy strategy preserves locality: if all references to an object are localized within the directory tree, then the other nodes need never create a directory entry for that object.

So far, we have considered only exclusive access to objects. Many applications would benefit from support for shared (read-only) access as well. We now outline a simple extension of the arrow directory protocol to support read-only replication. When multiple read-only copies exist, one is designated the *primary*. Each directory entry consists of a *primary link* pointing toward the primary copy, and a set of *secondary links*, each pointing toward one or more copies. When a node requests a read-only copy of an object, it follows any secondary link from each vertex, creating a new secondary link pointing back to itself at each node it visits. When a node requests exclusive access to the object, it first follows the primary links, flipping each primary link toward itself. When the node has acquired exclusive access to the primary, it *invalidates* the read-only copies by following, flipping, and consolidating the secondary links. When all invalidations are complete, the node has acquired the object. This protocol is *linearizable* [11]. More efficient protocols might be achieved by replacing linearizability with weaker notions of correctness [1].

We have not addressed the issue of fault-tolerance. Distributed mutual exclusion algorithms address the problem by "token regeneration", effectively electing a node to hold the new token. Here, the problem is different, since we cannot elect a new object, but we do need to recover navigational information that may be lost following a crash.

6 Related Work

Distributed directory management is closely related to the problem of distributed mutual exclusion. Naïmi, Tréhel, and Arnold [18] describe a distributed mutual exclusion algorithm (the *NTA* algorithm) also based on a dynamically changing distributed directed graph. In the NTA algorithm, when a node receives a message, it flips its edge to point to the node from which the request originated. In our algorithm, however, the node flips its edge to point to the immediate source of the message. As a result, our graph is always a directed subgraph of a fixed minimal spanning tree, while the NTA graph is more fluid.

The safety and liveness arguments for NTA and for our algorithm are similar, (though not identical), but the complexity models are quite different. The complexity analysis given in [18] is based on a model in which the underlying network is a clique, with unit cost to send a message from any node to any other. All nodes are assumed equally likely to request the token, and requests are assumed to occur sequentially. Under these assumptions, NTA shows that the average number of messages sent over a long period converges to $O(\log n)$. (Ginat [8] gives an amortized analysis of the NTA algorithm and some variants.) Our analysis, by contrast, models the underlying network as a weighted graph, where the end-to-end cost of sending a message is the sum of weights along its path. Instead of taking an average over a presumed uniform distribution, we compete against a directory with perfect instantaneous knowledge of the object's current location and no synchronization costs. The complexity analysis of Naïmi *et al.* does not apply to our algorithm, nor ours to theirs.

The NTA complexity model is appropriate for the general problem of distributed mutual exclusion, while ours is intended for the more specific problem of managing distributed directories in the Internet. Each link in our spanning tree is intended to represent a direct network link between routers, which is why our algorithm does not flip edges to point to arbitrarily distant network nodes (even though the routing service renders any pair of nodes "adjacent" for communication). The NTA complexity model, while mathematically quite elegant, does not seem well-suited for distributed Internet directory management, because point-to-point communication cost is not uniform in real networks, and access requests to distributed objects are typically not generated uniformly at random, but often have strong locality properties. For these reasons, a competitive complexity model seems more appropriate for our purposes.

Another mutual exclusion protocol employing path reversal is attributed to Schönhage by Lynch and Tuttle [16]. In this protocol, the directory is a directed acyclic graph. *Users* residing at leaf nodes request the object, and requests are granted by *arbiters* residing at internal nodes. Edges change labels and orientation to reflect the object's current status and location.

The arrow directory protocol was motivated by emerging *active network* technology [14], in which programmable network switches are used to implement customized protocols, such as application-specific packet routing. Active networks are intended to ensure that the cost of routing messages through the directory tree is comparable to the cost of routing messages directly through the network.

Small-scale multiprocessors (e.g., [9]) typically rely on broadcast-based protocols to locate objects in a distributed system of caches. Existing large-scale multiprocessors and existing distributed shared memory systems are either home-based, or use a combination of home-based and forwarding pointers [4, 5, 6, 12, 13, 15, 19].

Plaxton et al. [22] give a randomized directory scheme for read-only objects. Peleg [20] and Awerbuch and Peleg [3] describe directory services organized as a hierarchical system of subdirectories based on sparse network covers [2]. The earlier paper [20] proposed the notion of *stretch* to evaluate the performance of distributed directory services. In the worst case, these directories use per-object memory logarithmic in n, while ours is linear. (As described above, we believe we can avoid worst-case memory consumption in practice.) Strictly speaking, the asymptotic communication complexities are incomparable. Their notion of an optimal concurrent *find* execution is less conservative than ours, but for simplicity we will treat them as equivalent. The ratio of their concurrent *find* and *move* operations to their notion of an optimal protocol is polylogarithmic in n, while the ratio of our concurrent *find* and *move* operations to our (more conservative) notion of an optimal protocol is $(1 + \mathit{MST\text{-}stretch}(G))/2$. The arrow directory has lower asymptotic communication complexity when the MST-stretch of the network is less than some polylogarithm in n. Nevertheless, we feel the most important distinction is that the arrow directory protocol is much simpler than any of the hierarchical protocols, because it is explicitly designed to be implemented. We plan to report experimental results in the near future.

References

1. S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical Report ECE TR 9512 and Western Research Laboratory Research Report 95/7, Rice University ECE, Houston, TX, September 1995. A version of this paper appears in IEEE Computer, December 1996, 66-7.
2. B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Fast distributed network decompositions and covers. *Journal of Parallel and Distributed Computing*, 39(2):105–114, 15 December 1996.
3. B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the ACM*, 42(5):1021–1058, September 1995.
4. B. Bershad, M. Zekauskas, and W.A. Sawdon. The Midway distributed shared memory system. In *Proceedings of 38th IEEE Computer Society International Conference*, pages 528–537, February 1993.
5. J.B. Carter, J.K. Bennet, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 152–164, October 1991.
6. D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings Of The 4th International Conference on Architectural Support for Programming Langauges and Operating Systems*, pages 224–234. ACM, April 1991.
7. P. Fraigniaud and C. Gavoille. Memory requirement for universal routing schemes. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 223–243. acm, August 1995.
8. D. Ginat. Adaptive ordering of condending processes in distributed systems. Technical Report CS-TR-2335, University of Maryland, Computer Science, October 89.
9. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.
10. M.P. Herlihy. The Aleph toolkit: Platform-independent distributed shared memory (preliminary report). `www.cs.brown.edu/~mph/aleph`.
11. M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions On Programming Languages and Systems*, 12(3):463–492, July 1990.
12. K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 213–228, December 1995.
13. P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.
14. U. Legedza, D. Wetherhall, and J. Guttag. Improving the performance of distributed applications using active networks. Submitted to IEEE INFOCOMM, San Francisco, April 1998.
15. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1987.
16. N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TM-387, MIT Laboratory For Computer Science, April 1987.
17. N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory For Computer Science, November 1988.

18. M. Naïmi, M. Tréhel, and A. Arnold. A log(n) distributed mutual exclusion algorithm based on path reveral. *Journal of Parallel and Distributed Computing*, 34:1–13, 1996.
19. R. S. Nikhil. Cid: A Parallel, "Shared Memory" C for Distributed-Memory Machines. In *Proc. of the 7th Int'l Workshop on Languages and Compilers for Parallel Computing*, August 1994.
20. D. Peleg. Distance-dependent distributed directories. *Information and Computation*, 103(2):270–298, April 1993.
21. D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36:43–52, July 1989.
22. C.G. Plaxton, R. Rajaman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–321, June 1997.

This article was processed using the LaTeX macro package with LLNCS style