A Tale of Two Directories: Implementing Distributed Shared Objects in Java

Maurice Herlihy
Computer Science Department
Brown University
Providence, RI 02912
herlihy@cs.brown.edu

Michael P. Warres
Sun Microsystems
mpw@east.sun.com

November 28, 1999

Abstract

A *directory* service keeps track of the location and status of mobile objects in a distributed system. This paper describes our experience implementing two distributed directory protocols as part of the *Aleph toolkit*, a distributed shared object system implemented in Java. One protocol is a conventional *home*-based protocol, in which a fixed node keeps track of the object's location and status. The other is a novel *Arrow* protocol, based on a simple path-reversal algorithm.

We were surprised to discover that the Arrow protocol outperformed the home protocol, sometimes substantially, across a range of system sizes. This paper describes a series of experiments testing whether the discrepancy is due to an artifact of the Java run-time system (such as differences in thread management or object serialization costs), or whether it is something inherent in the protocols themselves. In the end, we use insights gained from these experimental results to design a new directory protocol that combines advantages of both.

A preliminary version of this paper appeared in the 1999 Java Grande Conference [1].

1 Introduction

Many distributed systems support some form of *mobile object*, which could be a file, a process, or any other data structure. A *directory* service allows nodes to keep track of mobile objects. In this paper, we are primarily interested in directories that track cached copies of shared data objects, but the techniques described could also be applied to other kinds of mobile objects.

This paper describes our experience implementing two distributed directory protocols as part of the *Aleph Toolkit* [14], a distributed shared object system under development at Brown. In the Aleph toolkit, as described below, a collection of *processing elements* (PEs), each a Java virtual machine, share a collection of *global objects*. Copies of global objects (either exclusive or shared) are cached at PEs on demand. The *directory manager* package is in charge of keeping track of each global object's cached copies, invalidating or moving them around as needed.

We consider two alternative directory protocols.

- The *Home* directory protocol is a conventional invalidation-based scheme in which each global object is associated with a fixed *home* PE. The home keeps track of the status and location of the object's cached copies.
- The *Arrow* directory protocol [12] is a recently-developed protocol based on a simple path-reversal algorithm.

This paper gives the first experimental comparison of the Arrow and Home directory protocols.

The Arrow protocol was originally conceived as a way to circumvent scalability problems inherent in home-based protocols, and a theoretical analysis appears elsewhere [12]. Nevertheless, we were curious how it would perform in practice on a relatively small-scale system, so we ran some simple distributed benchmarks to compare the Arrow protocol against the conventional Home directory protocol. To our surprise, the Arrow protocol outperformed the Home protocol, sometimes substantially, across a range of system sizes, and across different platforms and JDKs.

This paper describes a series of experiments devised to explain this discrepancy. One hypothesis is that the difference is an artifact of the Java run-time system, due to different ways in which the protocols make use of potentially expensive Java constructs such as threading, object serialization, transport-level protocols (e.g., RMI vs. UDP), etc. An alternative hypothesis is that the difference reflects inherent properties of the two protocols. In the end, we use insights gained from these experimental results to devise a novel *Hybrid* protocol that combines advantages of both the Home and Arrow protocols.

2 Background: The Aleph Toolkit

The work described here was performed in the context of the *Aleph toolkit* [14], a collection of Java packages that implements a platform-independent distributed shared object system. A distributed program runs on a number of logical processors, called *Processing Elements* (PEs). Each PE is a Java Virtual Machine, with its own address space. Aleph provides the ability to start threads on remote processors, and to communicate either by shared objects (with transparent synchronization and caching), or by message-passing or by ordered reliable multicast.

Structuring a distributed system as a toolkit allows programmers to "mix-and-match" different implementations of run-time system components without the need to restructure the application each time. To this end, the Aleph Toolkit isolates the most performance-critical and application-dependent components of the run-time system as distinct packages that can be replaced without having to restructure higher-level applications. Within the toolkit, each of these packages is accessible only through a Java *interface*, a language construct that constrains method signatures (and indirectly functionality). The Aleph toolkit provides one or more *default* implementations of these packages, and users are encouraged to substitute their own customized implementations. When a PE is started, it chooses at run-time which implementation to use based on run-time flags or a configuration file.

Aleph encapsulates a variety of run-time services behind interfaces, but the two that concern us here are the *directory manager*, in charge of managing replicated copies of distributed shared objects, and the *communication manager*, in charge of point-to-point message-passing. In this paper, we compare two distinct implementations of the directory manager, one based on the Home protocol, and one based on the Arrow protocol.

Within each implementation of the directory manager, point-to-point message-passing is handled by calls to the communication manager. Messages in Aleph are modeled loosely on *active messages* [28]. Each message encompasses a method and its arguments, and that method is called when the message is received. The abstract class `aleph.Message` implements `Serializable` and `Runnable`. A new message class is defined by extending `aleph.Message`, providing a `void run()` method to be called by the receiver. Messages sent from one PE to another are received in FIFO order. To minimize thread management overhead, if the programmer indicates that a message's `run()` method cannot block (i.e., pause for an unpredictable duration), then that method is run to completion before accepting the next message. Otherwise, the PE starts a thread to execute the `run()` method, and immediately accepts the next message. We currently have three implementations of the communication manager interface, based on TCP streams, UDP datagrams, and Java RMI, respectively.

3 Directory Protocols

Because PEs are distinct Java virtual machines, they cannot share regular Java objects. Aleph provides a `GlobalObject` class that allows PEs to share any serializable object. The code fragment in Figure 1 shows how to create a global object, open it, modify it, and commit the change. The methods of the `GlobalObject` class do little more than call the directory manager.

We now describe the two directory protocols (and their variants). For brevity, we omit many details. We describe each directory as if it were tracking the location of a single object, and we focus for now on exclusive access.

```
GlobalObject g = new GlobalObject(new Queue());
Queue q = (Queue) g.open("w");    // acquire exclusive access
q.enq(x);                         // modify the object
g.release();                      // commit and release
```

Figure 1: How to Use a Global Object

message	function	blocking?
`RetrieveRequest`	ask home for object	yes
`RetrieveResponse`	grant object to PE	no
`ReleaseRequest`	invalidate cached copy	yes
`ReleaseResponse`	confirm invalidation	no

Figure 2: Home Directory Messages

3.1 The Home Directory

The *home* directory is a "vanilla" scheme in which each global object is associated with a fixed PE, termed that object's "home". The home keeps track of the number, status and location of all cached copies of that object. There may be one read/write cached copy, or there may be multiple read-only cached copies. If a client has a cached copy of the object, it keeps track of whether the object is *busy* (in use by a local thread), and if so, whether the home has requested the copy to be returned or invalidated.

We will illustrate this protocol by tracing how one PE can acquire exclusive access to a global object held by another. When the client requests exclusive access to a global object, it does the following.

1. It sends a `RetrieveRequest` message to the object's home.
2. The home sends a `ReleaseRequest` messages to the PE holding the cached copy, and `RetrieveRequest.run()` blocks.
3. At the PE holding the copy, the `ReleaseRequest.run()` method blocks while the cached copy is in use. When object becomes free, the method invalidates the copy and returns a `ReleaseResponse` message to the home as confirmation.
4. At the home, the blocked `RetrieveRequest.run()` is notified, and it sends a `RetrieveResponse` message containing the current object copy to the requesting client.

Overall, acquiring exclusive access to an object requires four messages (two blocking). The other cases (e.g., read access) are similar. Figure 2 shows a table of message classes used by the home directory.

The Home directory protocol is simple and easy to implement, and similar schemes are widely used in distributed shared memory (DSM) systems and

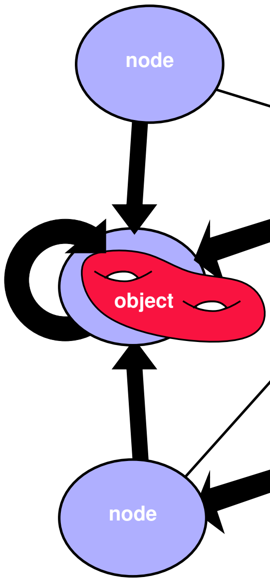

Figure 3: Initial Directory State

shared-memory multiprocessors ([8, 9, 10, 16, 17, 20, 23]. Nevertheless, any scheme in which an object has a fixed home suffers from potential problems of scalability and locality. As the number of PEs grows, or if an object becomes 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 PE currently holding the object is nearby.

3.2 The Arrow Directory

We start with the simplest form of the protocol, postponing refinements until later. The *arrow directory* is a tree spanning all PEs. Each PE stores a directory *link* in the form of an “arrow” which can point either to itself, or to any of its neighbors in the tree. The meaning of the link is the following: if a PE's link points to itself, then the object either resides at that PE, or will soon reside there. Otherwise, if the PE's link points to a neighbor, then the object currently resides in the component of the directory tree containing that neighbor. Informally, except for the PE that currently holds an object, a PE knows only in which “direction” that object lies.

The entire directory protocol can be described in a single paragraph. The directory tree is initialized so that following the links from any PE leads to the PE where the object resides (Figure 3). To acquire exclusive access to the object, a PE sends a `Find` message to the PE indicated by its arrow, and “flips” its arrow to point to itself (Figure 4). When a PE P whose arrow points to Q

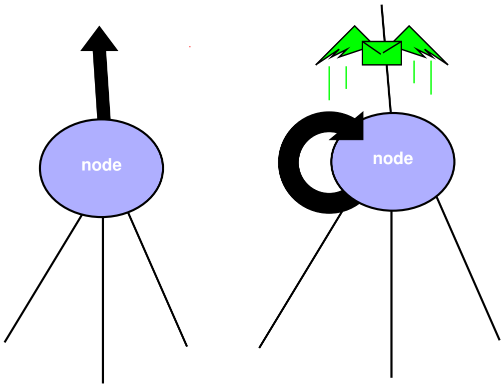

Figure 4: Path Reversal at Source Node

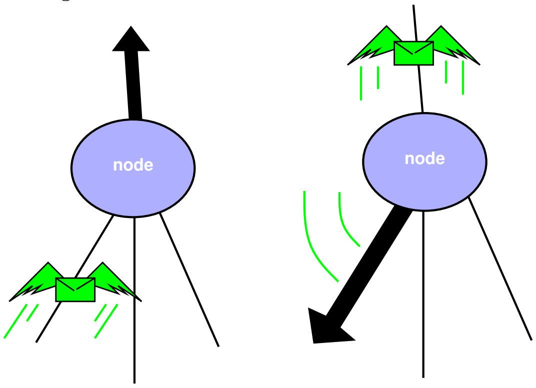

Figure 5: Path Reversal at Intermediate Node

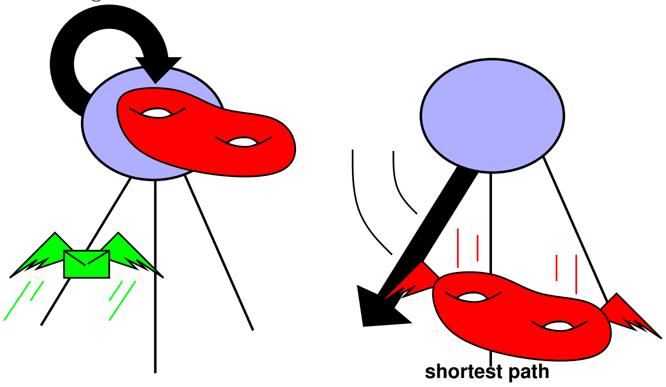

Figure 6: Path Reversal at Destination Node

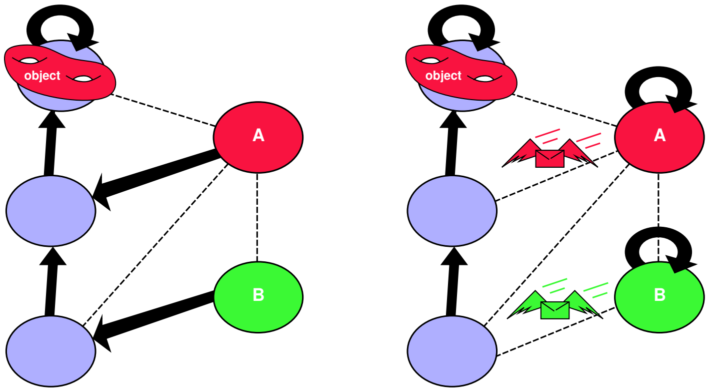

Figure 7: Two Find requests issued concurrently

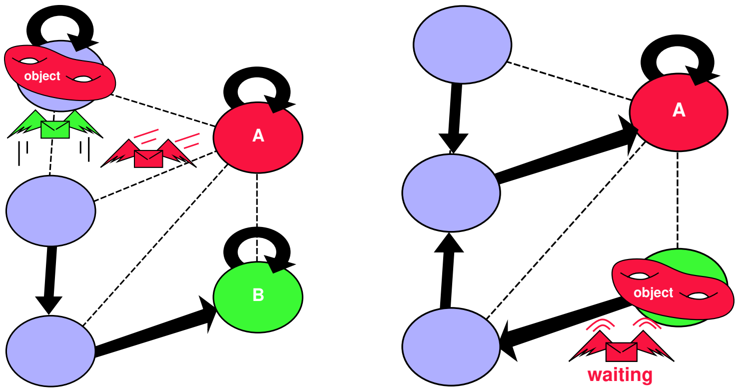

Figure 8: B's request overtakes and diverts A's request

receives a `Find` message from R, it immediately "flips" its arrow back to R. If it does not have a cached copy of the object, it forwards the message to Q, the prior target of its arrow (Figure 5). If it does have a copy, it waits until the copy is no longer in use, and sends the object back to the originator of the `Find` message. Note that the PE can send the object back directly, without further interaction with the directory (Figure 6).

An interesting aspect of the Arrow protocol is that it integrates synchronization and navigation in a single mechanism. Figure 7 shows a directory in which nodes A and B have issued concurrent `Find` requests. In Figure 8, B's request arrives first, and "diverts" A's request back to B. In this way, path reversal imposes a distributed queue structure on blocked `Find` requests, where each `Find` is buffered at its predecessor's PE. When a PE releases the object, it goes directly to the next waiting PE. This distributed queue structure is attractive for several reasons. First, it ensures that no single PE becomes a synchronization bottleneck. Second, if there is contention for an object, then each time that

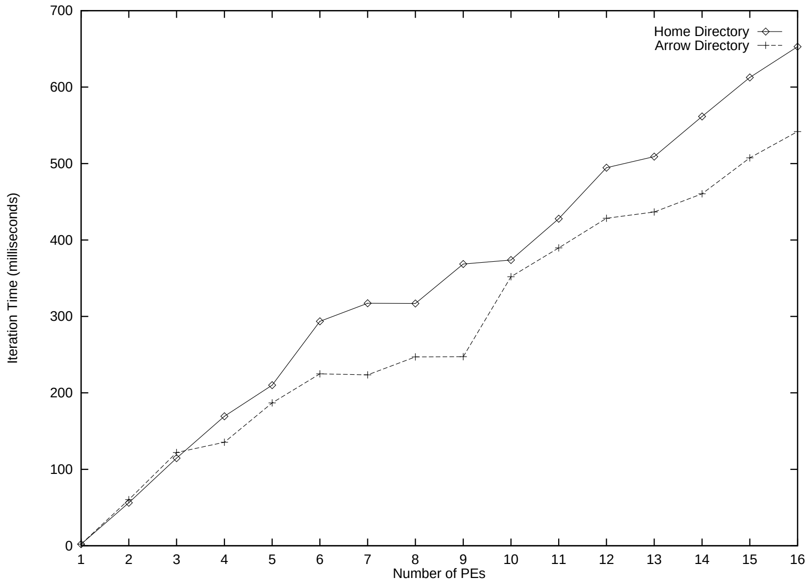

Figure 9: Comparison of arrow and home directories

object is released, that PE will already have a buffered `Find` request. In the limit, the latency 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). Finally, the queue structure ensures locality: each `Find` message takes a direct path through the directory tree to its predecessor's PE, with no detour to a home PE.

4 Experiments

The Arrow directory protocol was developed as an exercise in alleviating the scalability problems inherent in the home directory protocol [12]. We were curious, however, how the two protocols compare in practice on relatively small systems (sixteen or fewer PEs), so we implemented two versions of the Aleph directory manager, one using the home directory, and one using the arrow directory. We then ran some simple benchmarks to compare their performance.

We begin by describing the results of the simplest such benchmark, which measures the time needed for a collection of PEs to increment a shared counter. When the program starts at PE 0, it creates and initializes the counter. The program's main thread then proceeds in a sequence of rounds. In each round, the main thread starts a timer, and creates and starts a remote thread at each PE. Each thread then opens the shared counter (causing it to be moved into the local cache), increments it, and releases it. When all threads have completed their increments, the round ends, and the main program stops the timer. The entire process is repeated one hundred times, and the average delay across all rounds is recorded.

Times were measured in milliseconds by calls to `System.currentTimeMillis()`. All programs were executed using the local system's default just-in-time compiler. We ran tests on Sun workstations running Solaris 5.6 with JDK1.1.7A and JDK1.2, as well as Alpha workstations running Digital Unix V 4.0 and JDK1.1.8. The Sun workstations were linked to each other via 100MB Ethernet, while the Alpha workstations were connected by a mixture of both 10MB and 100MB networks. In this paper, we focus on the Solaris numbers under JDK1.2, but all the systems we observed behaved in essentially the same way.

We focus on the shared counter benchmark here because it is simple enough that performance differences are likely to reflect properties of the underlying directory protocols. This benchmark is not compute-bound, nor does it use multithreading. It tests the directories under fairly heavy contention, highlighting inherent performance differences.

Our initial comparison of the two protocols appears in Figure 9. To our surprise, we found that the Arrow directory outperformed the Home directory by a significant margin, across both small and medium sized PE groups. In the remainder of this paper, we explore the reasons behind this disparity. In particular, we were interested in determining whether the difference is an artifact of the Java implementation, or whether it is a result of the differences in the underlying algorithms.

4.1 Possible Java Artifacts

As shown in Figure 2, if the requested object is not immediately available, `RetrieveRequest.run()` blocks at the home PE. The object may be locked by another thread at the same PE, or it may be locked by a remote PE. When contention for an object is high, many threads may be blocked at the home waiting for the object to become available. When the object does becomes available at the home, all blocked `run()` methods are awakened by a call to `notifyAll()`. Although use of `notifyAll()` (instead of `notify()`) is considered good programming practice for safety reasons [18], it can result in a "thundering herd" effect if there are too many threads. Only one thread can successfully claim the object, but every blocked thread is awakened, checks the object's status, and returns to a blocked state. This threading overhead could conceivably cause the home directory to lag behind the Arrow directory in performance, since the Arrow directory does not use as many threads.

We tested this hypothesis in two ways. First, we instrumented the home directory to collect threading and wait queue statistics. The results indicate that the performance disparity between the Home and Arrow directories is not thread-related. Second, we reimplemented the home directory so that a message whose `run()` method would block places itself on a linked list instead. Even so, the modified version still ran significantly slower than the Arrow directory implementation.

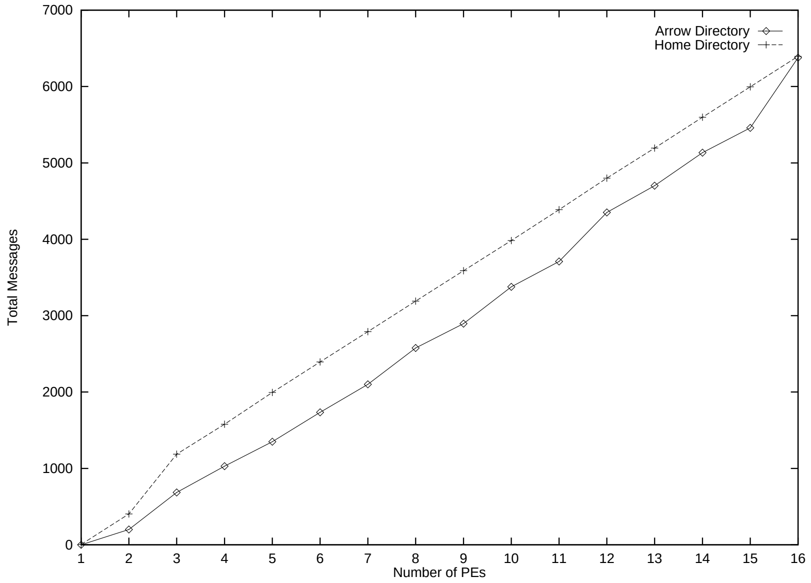

Figure 10: Number of Messages

Another hypothesis is that the performance lag was somehow related to features of the underlying communication manager. To test this hypothesis, we ran the benchmarks using two distinct communication manager implementations: one based on Java RMI and one using UDP datagrams. Although both Home and Arrow directory protocols performed faster with the RMI communications manager, the home directory protocol was still comparatively slower under both JDK 1.1.7 and JDK 1.2. Repeating the tests on Digital Unix instead of Solaris yielded equivalent results.

Another hypothesis is that the benchmark results were somehow skewed by the costs of object serialization (reputed to be expensive). Replacing the built-in serialization methods associated with the `Serializable` interface with hand-crafted serialization methods associated with the `Externalizable` interface made both benchmarks run faster, but did not noticeably affect their relative performance.

A final hypothesis concerns favorable race conditions. We were concerned that the order in which threads were created, or some similar recurring race condition, might cause the object to traverse the Arrow tree in a particularly advantageous order, perhaps repeatedly passing the object from one PE to a neighbor. To confound such effects, we reimplemented the benchmark to start threads in a random order. This change had no effect on performance.

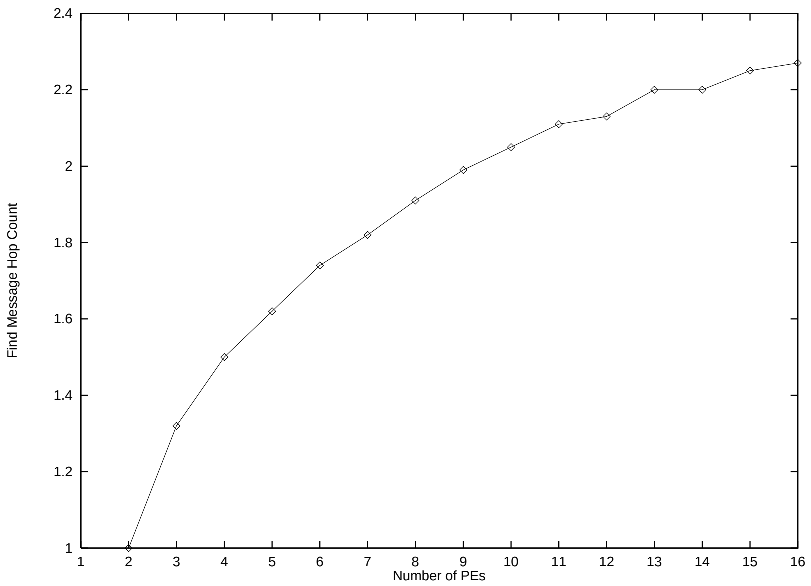

Figure 11: Number of Hops

4.2 Algorithmic Effects

Having eliminated the most plausible Java effects, we turned our attention to the protocols themselves.

How many messages do the protocols send? It turns out to be useful to distinguish between the message traffic needed to *request* an object, and the message traffic needed to *retrieve* an object.

The home directory usually requires two messages to request an object, one from the originating PE to the home, and one from the home to the PE currently holding the object. The Arrow directory, by contrast, requires d messages, where d is the distance from the requesting PE to the object's current location. Since our Arrow protocol implementation arranges PEs in a binary tree, the longest possible distance is 7 in a system of 16 PEs (roughly $2 \log n$ for n PEs). One might therefore expect the Arrow protocol to have significantly higher message traffic. In fact, it does not. As shown in Figure 10, the Arrow directory generates uniformly lower message counts. One reason is that the counter benchmark produces enough contention to ensure that most Arrow messages are quickly "diverted" to a neighboring PE, where they queue up awaiting the object. As a result, the actual message traffic under the Arrow protocol is substantially lighter than one might expect. Figure 11 counts the average number of hops taken by `Find` messages: even for 16 PEs, the average barely exceeds two. In summary, the observed message traffic needed to request an object is comparable in both protocols, although the potential traffic is higher in the Arrow protocol.

The major performance difference between the two protocols concerns the

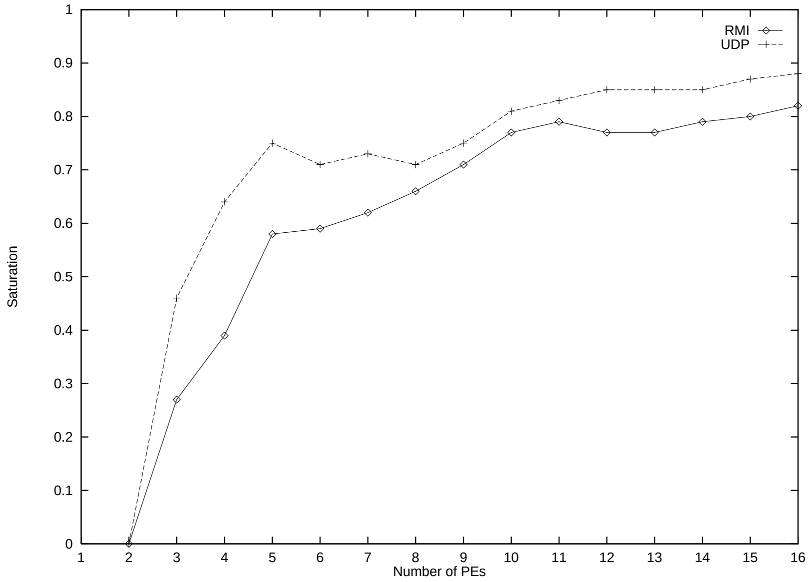

Figure 12: Saturation in Arrow protocol

way objects are retrieved. In both protocols, even at moderate levels of contention, whenever a PE releases an object, there is a waiting request for that object. The object request latency is increasingly hidden by the alternation of local computation and object retrieval. For this reason, the latency of requesting an object becomes less and less important as contention grows, and the latency of retrieving an object becomes more and more important. In the limit, the protocol alternates local computation and object delivery. Figure 12 shows the percentage of times an object is released to an already waiting request. The principal reason the Arrow directory is faster is that it requires only one message to retrieve an object, while the home protocol requires two.

4.3 Details

We now discuss some details of our Arrow directory implementation. Each of n PEs is given a unique index in the range 0 to $n-1$. The directory is organized as a binary tree, in a way that each PE's location in the tree can be calculated from its index. Since our experiments were conducted on workstations within the same local area network, we did not attempt to make the directory spanning tree reflect the underlying network topology.

The directory is initiated in the following "lazy" manner. When a PE creates a global object, it places a link in its local directory. Each remote reference to a global object includes the identity of the PE at which it was created. When a PE opens a global object, it consults its local directory. If the object is unknown, then the directory manager acts as if the arrow pointed toward the neighbor

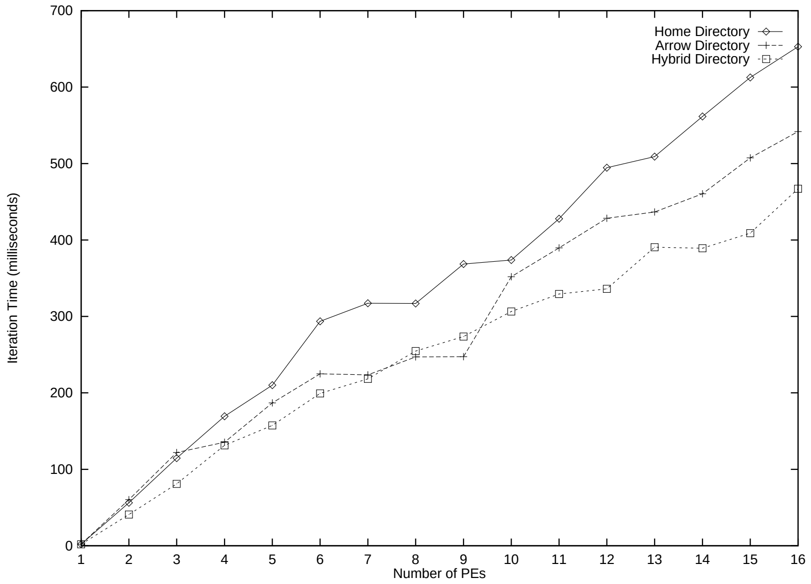

Figure 13: Comparison of home, arrow, and hybrid protocols

closer in the tree to the creating PE. In this way, the `Find` message finds its way to the object's creating PE, or to another PE that has a link for that object.

Read sharing is accomplished as follows. When a PE requests an object in read mode, the request is handled as described above. When a PE holding an object in read mode receives a `Find` message from another PE, it sends (a copy of) the object back immediately, without waiting for it to be released. This technique permits different PEs to read an object concurrently. and it does not require invalidation messages. This technique does not compromise sequential consistency. The principal limitation of this technique is that it cannot be extended to multi-step atomic transactions, since it violates the two-phase locking discipline by (effectively) releasing read locks before transaction commitment. Support for multi-step atomic transactions would require adding explicit invalidation messages.

So far, we have considered only synchronization among distinct PEs. Synchronization among threads within a single PE is handled by read (shared) and write (exclusive) locks implemented in the usual way. Each PE keeps a count of the threads that have an object locked. An object is released only when the count reaches zero. To minimize message traffic, local requests take priority over remote requests.

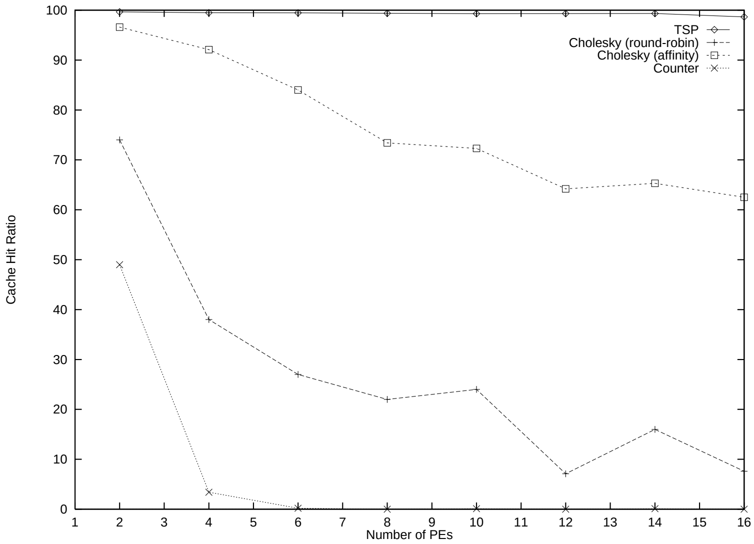

Figure 14: Cache hit ratios (Arrow protocol)

5 A Hybrid Protocol

The solution to our mystery can be summarized as follows. Contrary to our original expectations, the performance disparity between the home directory and the Arrow directory is not an artifact of the Java run-time system, it is inherent in the protocols themselves. The home directory is (in principle) more efficient at requesting an object, but the Arrow directory is more efficient at retrieving it. When there is contention for the object, however, request latencies are hidden by computation and retrieval latencies, so the retrieval latency dominates.

Motivated by this observation, we designed a simple *hybrid* protocol combining the best aspects of the home and Arrow protocols. Each object has a *home* PE, but the home tracks only the last PE to request access to the object. A PE requests an object by sending a message to its home, which forwards the message to the last PE to request that object. Requesting PEs are linked in a chain just as in the Arrow protocol. As a result, it takes two messages to request an object, and one to retrieve it. On the other hand, the Hybrid protocol is potentially vulnerable to hot-spot effects, since the object's home remains a potential bottleneck. Concurrent read access is handled as in the Arrow protocol, releasing read-only objects immediately on demand.

As shown in Figure 13, the hybrid protocol outperforms both the home and the Arrow protocol. Similar results were observed on the more complex benchmarks considered below in Section 6.

6 Other Benchmarks

As discussed above, the simplicity of the counter benchmarks makes it easy to isolate algorithmic effects. In this section, we consider three additional benchmarks: two implementations of parallel Cholesky decomposition, and a parallel traveling salesperson search. *Cholesky* decomposition is a parallel matrix decomposition algorithm, organized as a sequence of phases separated by barriers. In each phase, each PE is assigned a list of matrix columns to update. Updating a column typically requires reading other columns. In the *round-robin* implementation, work is assigned to PEs in a round-robin order, ensuring that each PE receives the same amount of work in each phase. In the *affinity* implementation, each column is associated with a single PE, which is the only PE allowed to modify that column. When a column needs to be updated, its PE is requested to do the work. Finally, the *traveling salesperson* (TSP) benchmark performs a parallel search for an efficient path through a weighted graph.

A natural way to classify directory benchmarks is by the application's cache hit ratio: the higher the miss ratio, the harder the directory manager has to work. (In the limit, an application that never misses doesn't need a directory manager.) As illustrated in Figure 14, TSP has the highest hit ratio, followed by affinity Cholesky, round-robin Cholesky, and finally the Counter, which misses almost every time. When the hit ratio is very high, the three protocols perform pretty much the same, while the Arrow and Hybrid protocols become more effective as the hit ratio decreases.

Each measurement reported here is the average of ten successive runs, taken at a time when the system was not heavily loaded. The number of workstations tested ranged from two to sixteen (counting by two to save time). Benchmarks subject to direct comparison were run one right after the other to decrease the likelihood that transient loads on the network or workstations would skew any particular benchmark. The numbers reported here were taken using a network of Alpha workstations running Digital Unix, under JDK1.1.8, at the Compaq Cambridge Research Lab. The experiments were partially replicated on a network of Solaris workstations running JDK1.2.2, at the Brown Computer Science Department. For up to eight workstations, both environments yielded essentially the same results. Beyond eight workstations, we found that fluctuations in network congestion due to other ongoing activities made it difficult to make reproducible measurements in the Brown environment.

Performance appears to be correlated with the cache-miss ratio. Figure 15) shows the elapsed times for the round-robin Cholesky benchmark on a 30-by-30 matrix. This benchmark has a relatively high cache-miss ratio. Here, the Arrow directory has the overall best performance, closely followed by the Hybrid directory, with the Home protocol substantially behind. Figure 16) shows the elapsed times for the affinity Cholesky benchmark on a 90-by-90 matrix. This benchmark has a much lower cache-miss ratio. Here, the hybrid protocol performs best, while the Arrow and Home are more-or-less comparable. Figure 17 shows the elapsed time for the TSP benchmark on a randomly-generated 128-node graph. This benchmark has a very low cache-miss ratio. Here, the three

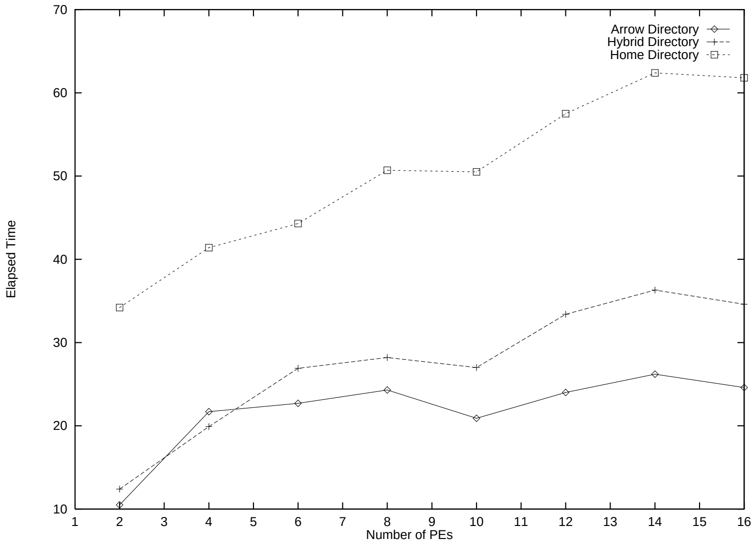

Figure 15: Cholesky (Round-Robin) Times

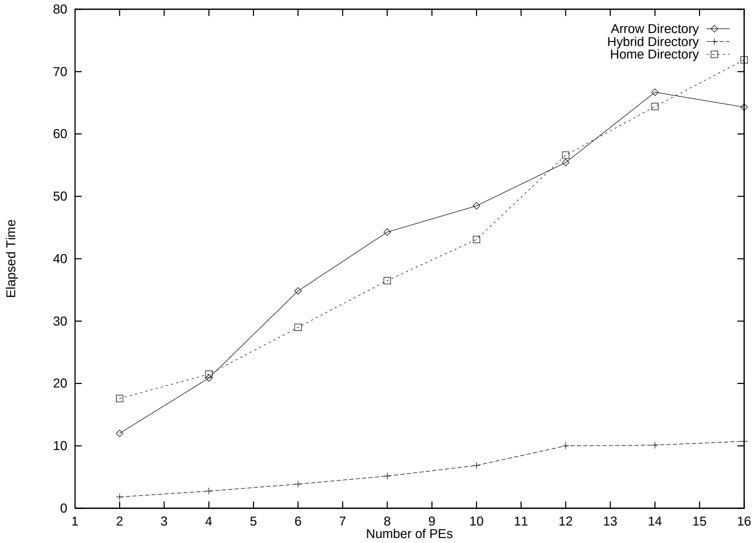

Figure 16: Cholesky (Affinity) Times

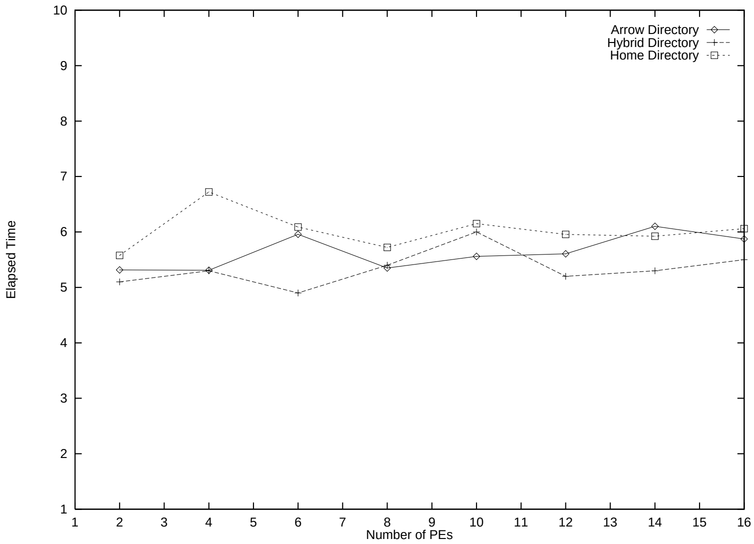

Figure 17: TSP Times

protocols are roughly comparable (although Hybrid still has a slight advantage).

In conclusion, the effectiveness of the Arrow and Hybrid protocols appears to be directly correlated with the cache-miss ratio: the greater the proportion of cache misses, the more effective the two path-reversal protocols.

7 Related Work

The home directory can be modified to allow an object's home to move. For example, Li and Hudak [20] 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.

Distributed mutual exclusion algorithms based on path reversal include Naïmi, Tréhel, and Arnold [22], Schönhage (as attributed by by Lynch and Tuttle [21]). As discussed in detail elsewhere [14], none of these algorithms uses path-reversal in the same way as the Arrow protocol.

The Arrow directory protocol was motivated by emerging *active network* technology [19], 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., [13]) typically rely on broadcast-based protocols to locate objects in a distributed system of caches. Existing large-scale multiprocessors and existing DSM systems are either home-based, or use a combination of home-based and forwarding pointers [8, 9, 10, 16, 17, 20, 23].

Plaxton et al. [26] give a randomized directory scheme for read-only objects. Peleg [24] and Awerbuch and Peleg [5] describe directory services organized as a hierarchical system of subdirectories based on sparse network covers [4]. A detailed analysis of the relative asymptotic complexity of these algorithms appears elsewhere [14].

Pioneering work on DSM systems includes Ivy [20], Munin [7], Treadmarks [17], Midway [8], and others. Early work on language support for DSM includes Linda [2] and Orca [6]. The early Aleph design was substantially influenced by experience using the Cid DSM system [23]. In Cid, as in CRL [16], an object is constrained to be a contiguous region of memory, a restriction not well-suited to languages such as C++ or Java where objects are typically implemented as non-contiguous list structures.

DSM projects based on Java include Java/DSM [29], and Mocha [27]. Java/DSM implements a parallel Java Virtual Machine (JVM) running on top of Treadmarks [17]. The Infospheres [11] project is also based on Java, but has less of an emphasis on shared objects. Mocha, like Aleph, provides the ability to run threads at different nodes, and to share objects among those threads, without modifications to the JVM. Mocha provides a substantially different API, with an emphasis on fault-tolerance and replication. The Jini system [3] provides Java-based support for distributed systems with a focus on "federating" distinct services. JavaParty [25] provides language support for remote objects and threads, while Kan [15] also supports nested transactions.

References

[1] ACM Java Grande Conference. *A Tale of Two Directories: Implementing Distributed Shared Objects in Java*, 1999.

[2] S. Ahuja, N. Carriero, and D. Gelernter. Linda and Friends. *IEEE Computer*, 19(8):26–34, August 1986.

[3] K. Arnold, B. O'Sullivan, R.W. Schiefler, J. Waldo, and A. Wollrath. *The Jini(tm) Specification.* Addison-Wesley, 1999.

[4] 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.

[5] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the ACM*, 42(5):1021–1058, September 1995.

[6] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with Distributed Programming in Orca. In *Proc. of the 1990 Int'l Conf. on Computer Languages*, pages 79–89, March 1990.

[7] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, March 1990.

[8] 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.

[9] 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.

[10] 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.

[11] K. M. Chandy, A.dam Rifkin, P. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and Luke Weisman. A world-wide distributed system using java and the internet. In *IEEE International Symposium on High Performance Distributed Computing (HPDC-5)*, August 1996.

[12] M. Demmer and M.P. Herlihy. The arrow directory protocol. In *Proceedings of 12th International Symposium on Distributed Computing*, September 1998.

[13] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–70, June 1990.

[14] M. Herlihy. The aleph toolkit: Support for scalable distributed shared objects. In *Proc. of the Third Int'l Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'99)*, January 1999.

[15] J. James and A. Singh. Design of the Kan distributed object syste. Technical Report TRCS99-29, Department of Computer Science, University of California at Santa Barbara, August 1999.

[16] 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 (SOSP-15)*, pages 213–228, December 1995.

[17] 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.

[18] Doug Lea. *Concurrent Programming in Java.* Addison Wesley, 1996.

[19] U. Legedza, D. Wetherhall, and J. Guttag. Improving the performance of distributed applications using active networks. IEEE INFOCOMM, San Francisco, April 1998.

[20] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1987.

[21] 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.

[22] 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.

[23] 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.

[24] D. Peleg. Distance-dependent distributed directories. *Information and Computation*, 103(2):270–298, April 1993.

[25] Michael Philippsen and Matthias Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[26] 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.

[27] B. Topol, M. Ahamad, and J.T. Stasko. Robust state sharing for wide area distributed applications. Technical Report GIT-CC-97-25, Georia Institute of Technology, Atlanta, GA, September 1997.

[28] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, May 1992.

[29] Weimin Yu and Alan Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.