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@ast.sun.com

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 runtime 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 usually outperforms both.

1 Introduction

Many distributed systems support some form of mobile object. A mobile object 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.

This paper describes our experience implementing two distributed directory protocols as part of the Aleph toolkit [14], a distributed shared object system implemented in Java. 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 [11] 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 [11]. 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 runtime 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-based and arrow protocols, and outperforms them both on a range of benchmarks.

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

Figure 1: How to Use a Global Object

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 [27]. 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. 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 two implementations of the communication manager interface, one based on UDP datagrams, and one based on Java RMI.

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.

<table>
<thead>
<tr>
<th>Message</th>
<th>Function</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>RetrieveRequest</td>
<td>ask home for object</td>
<td>yes</td>
</tr>
<tr>
<td>RetrieveResponse</td>
<td>grant object to PE</td>
<td>no</td>
</tr>
<tr>
<td>ReleaseRequest</td>
<td>invalidate cached copy</td>
<td>yes</td>
</tr>
<tr>
<td>ReleaseResponse</td>
<td>confirm invalidation</td>
<td>no</td>
</tr>
</tbody>
</table>

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, and we will not dwell on them. Figure 2 shows a table of message classes used by the home directory.

The home directory is simple and easy to implement, and similar schemes are widely used in distributed shared memory (DSM) systems and shared-memory multiprocessors [6, 8, 9, 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 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 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 entry 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. To acquire exclusive access to the object, a PE sends a Find message to the the PE indicated by its arrow, and "flips" its arrow to point to itself. When a PE $P$ whose arrow points to $Q$ 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. 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 3 shows a properly initialized directory, where directed edges correspond to the directory links. The object resides at PE $u$ (indicated by a square). When PE $v$ requests the object, it sends a Find to the node pointed to by its arrow, and then “flips” the arrow to itself. When $u$ receives the message from $v$, it forwards it to $u_2$, and flips its arrow back to $v$. Figure 4 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 $u_3$ points to $u_1$.

As shown in Figure 5, the Find message continues to flip links until it arrives at the PE where the object resides. When that PE receives the Find message, it responds by sending the object directly back to $v$. As illustrated in Figure 6, after $v$ has received the object, all links in the directory tree again lead to the object’s current location, so any subsequent Find messages will be directed there.

An interesting aspect of the arrow protocol is that it integrates synchronization and navigation in a single mechanism. When the arrow directory is in a quiescent state (no messages in transit), the directory 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 in order to each of the waiting PEs. This distributed queue structure is attractive for several reasons. First, it ensures that no single PE becomes a synchroniza-
tion bottleneck. Second, if there is contention for an object, then each time that 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

In principle, the arrow directory appears more scalable than the home directory [11], but we were curious how the two protocols compare in practice on relatively small systems (sixteen or fewer PEs), so we implemented two versions of the Alpha directory manager, one using the home directory, and one using the arrow directory. We then ran some simple synthetic benchmarks to test their relative performance.

We now describe the results of the simplest such benchmark, which measures the time needed to 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 once hundred times, and the average delay across all rounds is recorded.

Times were measured in milliseconds by calls to System.

currentT imeMillis(). All programs were executed using the default just-in-time compiler. We ran tests on Sun workstations running Solaris 5.6 with JDK 1.1.7A and JDK 1.2, as well as Alpha workstations running Digital Unix V 4.0 and JDK 1.1.6. 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 abstract, we focus on the Solaris numbers under JDK 1.2, 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 7. 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, RetrieveRequest.run() blocks at the home PE if the requested object is not immediately available. 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 re-implemented 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.

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 improved performance slightly (and equally) for both directories.

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.

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
Figure 7: Comparison of arrow and home directories

Figure 8: Number of Messages
Figure 9: Number of Hops

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 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. As shown in Figure 11, the hybrid protocol outperforms both the home and the arrow protocol.

6 Other Benchmarks

As described above, we focus on the counter benchmarks (and its variants) because its simplicity makes it easy to isolate algorithmic effects. Nevertheless, we also tested the directories under more complex benchmarks. We compared the Home directory, the Arrow directory, a version of the Arrow directory that also supports shared access, and the Hybrid directory, all running on eight PEs under JDK 1.2 on Solaris. Times shown are normalized to the running time under the Home directory.

Figure 12 shows the running time for parallel Cholesky decomposition, Figure 13 shows a parallel traveling salesman problem, and Figure 14 a parallel ray-tracing benchmark. In each benchmark, the addition of shared access speeds up the Arrow protocol by only a modest amount. The Hybrid protocol does poorly on the Traveling salesman benchmark, and reasonably well on the others.

7 Discussion

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,
Figure 10: Saturation in arrow protocol

Figure 11: Comparison of home, arrow, and hybrid protocols
Figure 12: Cholesky Benchmark: 8 PEs

Figure 13: Traveling Salesman Benchmark: 8 PEs

Figure 14: Ray Tracing Benchmark: 8 PEs
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 an entry in its local directory. Each remote reference to a global object includes the identity of the PE at which it was created. When 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 creating PE. In this way, the Find message finds its way to the object’s creating PE, or to another PE that has an entry for that object.

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.

We extended the Arrow directory protocol to support shared access in the following way. When a PE opens an object, it specifies shared or exclusive access. When a Find message requesting shared access arrives at a node holding an object in shared mode, the object is granted immediately, and the arrows are flipped in the usual way. When the object is sent to the new node, however, the message includes a list of PEs that hold the object in shared mode. Before another PE can acquire exclusive access to the object, it must invalidate all those shared copies.

8 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 Námi, Tréhel, and Arnold [22], Schönhofer (as attributed by by Lynch and Tuttle [21]). A comprehensive overview of such schemes is given by Helary et al. [13]. 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., [12]) 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 [6, 8, 9, 16, 17, 20, 23].

Plaxton et al. [25] give a randomized directory scheme for read-only objects. Peleg [24] and Awerbuch and Peleg [3] describe directory services organized as a hierarchical system of subdirectories based on sparse network covers [26]. A detailed analysis of the relative asymptotic complexity of these algorithms appears elsewhere [14].

Pioneering work on DSM systems includes Ivy [20], Munin [5], Treadmarks [17], Midway [7], and others. Early work on language support for DSM includes Linda [1] and Orenda [4]. 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 [26]. Java/DSM implements a parallel Java Virtual Machine (JVM) running on top of Treadmarks [17]. The Infospheres [10] 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 [28] provides Java-based support for distributed systems with a focus on “federating” distinct services. Java-Party [15] provides language support for remote objects and threads.

References


