The Aleph Toolkit: Support for Scalable Distributed Shared Objects

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

Abstract

The shared object model is an appealing programming abstraction for distributed computing. By hiding the details of the network and data distribution, it allows the programmer to focus on higher-level concerns, and makes the program structure robust in the presence of changes in distribution patterns or environment. Nevertheless, it is not at all clear that the distributed shared object model can be adapted to the needs of modern large-scale distributed applications.

The Aleph Toolkit is a collection of Java packages intended to support the construction of distributed shared objects in a way that addresses networking-related performance issues. This paper describes the design and rationale for the Aleph API, as well as our preliminary experience implementing a distributed shared object system in Java.

1 Introduction

The shared-object model is an attractive approach to structuring distributed applications. Existing shared-object systems, however, often lack the flexibility to meet the demands of large-scale networked applications. We believe that a toolkit approach is the most promising way to achieve an adequate level of flexibility and application-specific customization. Instead of providing a monolithic collection of services, a toolkit encapsulates individual services behind interfaces, allowing applications to select (or develop) customized implementations of each service.

The Aleph Toolkit is a collection of Java packages intended to support the construction of customized distributed shared objects. Aleph supports both "push-based" and "pull-based" communication, and "data-shipping" and "control-shipping" patterns. The major components of the Aleph run-time system are defined by Java interfaces, allowing programmers to substitute customized implementations that adapt to the needs of applications, or exploit specialized hardware.

This paper describes the overall design and rationale for the Aleph API and internals, as well as our early experience implementing a distributed shared object system in Java. The interesting aspects of this system include the modular decomposition, and the provision of services (such as ordered anonymous multicast) that differ from those provided by similar systems.

2 Programming Models

We are interested in scalable distributed applications that require rapid access to complex objects. Examples of such applications include electronic commerce and trading systems, traffic information systems, and interactive communication systems such as conferencing or whiteboards. Such applications often have a critical need to control the ways in which data objects are moved through the system. In an ideal world, an object's clients would have instant access to the object's current state. Often, however, it is not possible to provide clients with a completely accurate view of the object, thus the application must allow the client's views of the object to diverge from the ideal view in a controlled way.

We call this class of issues the *data fidelity problem*. The conventional way to implement distributed shared objects, in which cached copies are moved among clients on demand, is clearly inadequate to these challenges. Instead, application designers must determine an application-specific notion of fidelity: how closely clients' views must track the object's actual state.

*This work is supported by AFOSR Agreement F30602-96-2-0228, DARPA Order D885.

A simple stock trading example illustrates many of these issues. Each stock's price is represented as a distributed *quote* object, where each client accesses the object through a proxy. If there is a single server, the number of clients is small, and each client is interested in a different stock, then quotes could be kept up-to-date by having proxies periodically poll the server (a "pull-based" approach). If the number of clients is large, or the clients have similar interests, then it is more sensible for the server to multicast changes to quotes (a "push-based" approach). If certain clients want to be notified when certain stocks change by specified percentages, then it makes sense for the server to unicast updates to those clients' proxies. Even more elaborate schemes could be imagined: updates may be distributed among a hierarchy of caching servers, clients could be organized into multiple multicast groups depending on their interests, and so on.

Programming models based on explicit message passing (such as PVM [12] or MPI [11]) are widely used for scientific and engineering applications. Although message-passing has proved useful for relatively small-scale and regularly-structured applications, we believe it is poorly suited to structuring large-scale or long-lived applications. The principal limitation of the message-passing model is that it burdens the programmer with responsibility for all interprocessor communication, synchronization and caching. This burden is particularly cumbersome for large-scale applications or those with irregular or dynamically changing communication structures.

The work described here has been influenced by several kinds of distributed shared object systems. *Distributed shared memory* (DSM) systems, whether page-based or object-based, implement a "data-shipping" model, in which the bits representing the object are moved among client caches on demand. A variety of techniques have been proposed for maintaining consistency among the cached copies [1]. Many existing DSM systems provide only limited *flexibility*: although most DSM systems support a fixed number of cache coherence protocols, the stock quote example given above illustrates the need to define and implement application-specific notions of consistency that cannot be anticipated by the designers of the DSM system. A second limitation is that the data-shipping model is simply not appropriate for some kinds of shared objects, including objects that represent caches or encapsulate services, or objects whose contents are subject to security constraints. For example, consider a system that allows users to open chat windows on one another's displays. One might like to treat another user's display as an object providing methods to open and close chat windows, but such a display object could not be treated simply as a data structure to be paged across the network. Similar considerations apply to the system's authentication server, which encompasses sensitive information such as passwords.

By contrast, *remote method invocation* (RMI) systems [17, 19] implement a "control-shipping" model for objects. Calls to an object's methods are transformed, via a stub, to messages forwarded to a remote site that encapsulates the object's state. The RMI approach works well for objects that encapsulate services or resources, but additional techniques are needed to deal with objects for which flexible caching policies are important for performance.

Each of these choices, push vs. pull, or data-shipping vs. control-shipping, is sensible under certain circumstances. The range of choices suggests that a toolkit approach is an attractive way to allow object implementers to "mix-and-match" module implementations to meet application-specific requirements and to track changes in the underlying hardware, reducing or eliminating the need to restructure the application each time.

3 Aleph API

The *Aleph toolkit* is a collection of Java packages (JDK 1.1) that provides platform-independent support for distributed shared objects. Our emphasis is not merely on *portability*, *i.e.*, adapting code written for one platform to another, but on *interoperability*: the ability to run computations that span multiple heterogeneous platforms.

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. Each PE is created as part of a *PEgroup* (`aleph.PEGroup`)[1] When the PE group is started, the Aleph run-time system supervises a handshake protocol ensuring that each PE is initialized knowing the address of every other PE within its group. Any PE in a PE group can shut down the entire group. It is also possible to create *long-lived* PEs that do not belong to any group. Such long-lived PEs are useful for long-lived services and applications.

The next section gives an overview of the Aleph communication primitives. We focus on the meaning of these primitives, and on how their use affects program structure.

[1] A PE group is not a process group in the sense used by Isis [7] and related systems.

```
public class RegisteredObject {
  static Hashtable registry;    // register objects here
  static PE[] peers;            // all PEs
  public RegisteredObject(String label) {  // constructor
    RegisterMessage m = new RegisterMessage(label, this); // construct message
    for (int i = 0; i < peers.length; i++) // send it out
      m.send(peers[i]);
    ...
  }
  ...
  // Message defined by static inner class
  private static class RegisterMessage extends aleph.Message {
    String label; RegisteredObject object; // new data fields
    public RegisterMessage(String label,   // constructor
                           RegisteredObject object) {
      this.label = label; this.object = object;
    }
    public void run() {         // actually register the object
      registry.put(label, object);
    }
  }
}
```

Figure 1: Illustrating use of `aleph.Message`

3.1 Message Passing

Direct message-passing is the simplest form of communication between PEs. It is the basis for point-to-point (unicast) communication, and is used extensively in the Aleph internals. Messages in Aleph are modeled loosely on *active messages* [22]. Each message encompasses a method and its arguments, and that method is called when the message is received. We define an abstract class `aleph.Message` that implements `Serializable` and `Runnable`. A new message class is defined by extending `aleph.Message`, including a `void run()` method to be called by the receiver. When a message is received, its `run` method is executed to completion. Messages sent from one PE to another are received in FIFO order, and their `run` methods are executed in that order.

We say that a message is *blocking* if its `run` method could be blocked for an unpredictable duration. We emphasize that a message is not necessarily blocking just because it calls a `synchronized` method, because many `synchronized` methods impose only a bounded delay. For example, if P sends Q a message requesting exclusive access to an object, that message is blocking because an application running at Q may not be ready to release the object. When Q eventually sends a reply message containing that object to P, the reply would not be considered blocking, even if the reply's `run` method calls a synchronized method to place the object in a shared hash table, because that insertion cannot be blocked for an arbitrary duration.

Blocking messages belong to the class `aleph.AsynchMessage`, a subclass of (non-blocking) `aleph.Message`. When a PE receives a blocking message, it allocates a thread to execute its `run` method, and the PE immediately proceeds to receive the next message. When a PE receives a non-blocking message, it transfers the message to a dedicated thread that that executes successive messages' `run` methods to completion. This distinction between blocking and non-blocking messages yields a modest performance benefit by reducing reducing thread-switching overhead (See Figure 3). More importantly, however, it yields the semantic benefit of ensuring that non-blocking `run` methods are executed in FIFO order, a guarantee that can substantially simplify protocol design. (Since blocking messages' `run` methods are executed asynchronously in independent threads, they may effectively be executed out of order).

As a language construct, Messages provide a clean way for a class residing at one PE to communicate with instances of the same class at other PEs. Figure 1 shows a simple `RegisteredObject` class in which an object created at one PE is registered in a static table at all PEs. The message class is defined as a a static inner class, so its `run` method has direct access to class (static) variables, but the message class itself is not visible outside the class, and does not clutter up the class's specification.

3.2 Events

The `aleph.Event` class supports push-based communication. An event object is a kind of multicast channel. One PE can notify others when an event occurs by calling that `Event` object's `signal` method (with an optional argument). If a PE wants to be notified when an event has been signaled, it registers a `Listener` object with that event. Just as in the Abstract Window Toolkit (AWT), the listener is a dummy object that provides a `void actionPerformed(Object object)` method that is called (with the optional argument) when the event is signaled. If a PE loses interest in the event, it can unregister its listener.

Events provide a form of reliable anonymous multicast. The multicast is reliable because all PEs registered with an event will eventually be notified when the event is signaled. It is anonymous because a PE signaling an event need not know the identities of the other PEs listening to that event. We believe that anonymity is a precondition for scalability: registering or unregistering a listener should be a light-weight operation, analogous to joining or leaving an IP multicast group.

Events can be ordered or unordered. If an event is ordered, notifications are delivered to listeners in the same order. We use unordered events in barrier objects (many senders, many listeners), join objects (many senders, one listener). Ordered events are intended for push-based proxy or cache coherence protocols, in which PEs install incremental modifications to remote object proxies. Figure A simplified barrier using events appears in Figure 2.

There are two kinds of signals: *normal*, and *flush*. When a PE registers a listener with an event object, it is notified of all signals back to the most recent flush. The distinction between normal and flush signals is intended to allow the application to inform the Event implementation when it is safe to discard information about past notifications. For example, a PE might multicast incremental changes to an object using normal signals, but then periodically multicast the object's complete state using a flush.

Our notion of ordered multicast can be contrasted to that used in systems employing virtual synchrony [3, 7, 21]. In those systems, PEs belong to a *process group* whose membership is common knowledge among the group members. Each time a PE enters or leaves the group, the members undertake a kind of global consensus protocol to ensure that all current participants agree on the group membership. By contrast, Aleph Events are intended to permit PEs to enter or leave the multicast group with a minimum of disruption.

3.3 Remote Threads and Functions

A thread running within one PE can invoke a thread within another PE, and optionally wait for that thread to finish. For example, in the classic "hello world" application, the remote thread that will execute at each PE is defined as:

```
static class HelloThread
              extends aleph.RemoteThread {
  public void run() {
    System.out.println("Hello World from "
                        + PE.thisPE());
  }
}
```

Remote threads extend the abstract class `RemoteThread`. Like regular Java threads, the class must provide a public `void run()` method to be called when the thread is started.

As usual for Java programs, the top-level class must include a method with signature

```
public static void main(String[] args)
```

to be called when program starts. The `main` method creates an instance of a remote thread object.

```
HelloThread thread = new HelloThread();
```

As with regular threads, a remote thread does not execute until it is explicitly started. The `main` method then creates a `Join` object for synchronization, enumerates all PEs, starts an instance of `HelloThread` at each PE[2], and waits until all remote threads have completed.

```
Join join = new Join();
for (Enumeration e = PE.allPEs();
      e.hasMoreElements();)
  thread.start((PE) e.nextElement(), join);
join.waitFor();
```

There is also a `RemoteFunction` class that allows remote threads to return values.

[2]Because starting a `RemoteThread` actually starts a copy of the thread object, a single `RemoteThread` instance can be started more than once.

```
public class Barrier implements Serializable {
  private int count;            // number of outstanding calls
  private Event event;          // used for communication
  /**
   * Construct a barrier, one thread per PE.
   **/
  public Barrier() {
    this(PE.numPEs());
  }
  /**
   * Wait for all threads to reach this barrier.
   **/
  public synchronized void waitFor() {
    event.setListener(new Listener() { // anonymous class
      public void actionPerformed(Object object){
        synchronized (Barrier.this) {
          count -= 1;
          Barrier.this.notifyAll();
        }
      }});
    event.signal();
    while (count > 0) {
      try { wait(); } catch (InterruptedException e) {};
    }
  }
```

Figure 2: Illustrating use of `aleph.Event`

4 Toolkit Implementation

Programming in Java to standard APIs supports portability and interoperability among different operating systems. Even so, there is another important dimension to portability that remains a challenge: effectively and economically exploiting emerging hardware platforms for network switching, memory interconnection, and clustered computing. A sensible way to isolate applications from such shifts in hardware foundations is to identify the modules most likely to be affected by advances in technology, and to isolate each one behind a Java *interface*, a language construct that constrains method signatures (and implicitly constrains functionality). For each such module, the Aleph toolkit provides one or more *default* implementations. Users are encouraged to substitute their own customized implementations, especially "native" implementations that exploit specialized or exotic hardware. We now give a brief description of the principal packages.

- *Communication Manager* Transport-level communication within the Aleph toolkit is mediated by the *Communication Manager* interface. Aleph currently provides two Communication Manager implementations: one uses Java RMI (based on TCP stream sockets), and the other uses IP Datagrams. These packages are discussed further in the performance section below. We are about to acquire an ATM switch, and we are in the process of constructing a "native" implementation for that medium.

- *Directory Manager* The *Directory Manager* locates a shared object's proxies. Aleph currently provides two implementations of the directory manager interface: a conventional "home directory" scheme, in which each object has a home PE that keeps track its current location and status, and a novel "arrow" directory [9] scheme, based on a simple path reversal algorithm, which has better scalability properties. We consider directory manager implementations a rich source of future research.

- *Event Manager* Aleph Event implementation is encapsulated behind the *Event Manager* interface. Techniques for reliable anonymous multicast remain an active area of research (for example, [10, 15]). Most techniques for *ordered* multicast originate from the Virtual Synchrony community (for example, [3, 7, 21]), but the global synchronization needed to track group membership changes would defeat our goal of achieving scalability through anonymity. Several members of our research group are working on novel techniques for event implementation. In the meantime, the current Aleph release provides a simple Event Manager implementation that orders each event's signals via a "home" PE for that signal.

- *Transaction Manager* Aleph also provides support for atomic transactions encompassing threads that run within a single PE. We are working on a distributed transaction manager, as well as an integrated user-level checkpointing scheme.

Aleph also provides some support for instrumentation, and a registry service for long-lived PEs.

5 Performance

In this section, we examine the performance of the basic Aleph communication primitives. We will see that we can achieve reasonable communication latencies on a local area network, and that different implementations of the communication and directory managers provide different levels of performance.

Each time shown is a duration measured in milliseconds by calls to `System.currentTimeMillis()`, and each such duration is the average of one hundred successive tests. All programs were executed with the default just-in-time compiler. We ran tests on both Sun workstations running Solaris 5.6 and JDK1.1.7A, and Alpha Workstations running Digital Unix V 4.0 and JDK1.1.6. The workstations are linked by a complex mixture of 10MB and 100MB ethernets.

The first set of benchmarks measures round-trip times for simple messages, expressed as function of message length. The very first message-passing benchmark measures the latency of a null *remote method invocation* (RMI) from one host to another using the standard `java.rmi` package. This benchmark is not a test of the Aleph software. Instead, it is intended to establish a baseline for Java-based communication to which the Aleph implementation can be compared. This remote method takes an variably-sized object as an argument, and immediately returns its argument. The argument object includes an array of bytes whose size was varied from zero to 42000 in increments of 6000.

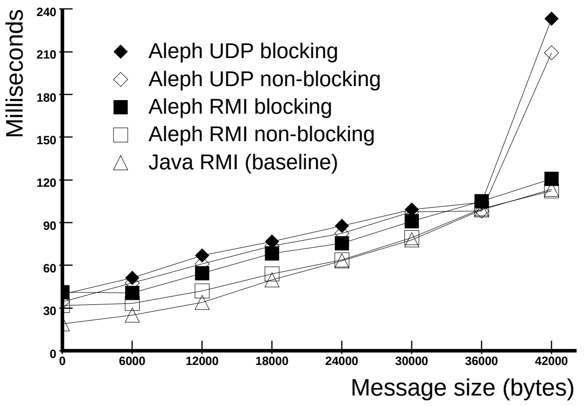

Figure 3: Round-Trip Message Latency (Digital Unix)

The remaining message-passing benchmarks tested round-trip times for Aleph messages. In each test, one PE sends a message to another. On delivery, that message's `run` method sends back a pre-allocated reply message. We measured the interval at the first PE between sending the first message and receiving the reply. Numbers shown are the average of 100 successive round trips. This interval encompasses the following steps: serializing, transmitting and deserializing the first message, executing its `run` method in a separate thread at the receiving PE, serializing, transmitting, and deserializing the reply message and executing its `run` method in a separate thread.

We tested two implementations of the Aleph `CommunicationManager` interface: one that uses Java RMI (TCP-based) and one that uses UDP datagram sockets. The results are shown in Figure 4 for two Digital Unix workstations. In the RMI-based communication manager, the thread that receives the message immediately passes a non-blocking message the message to a second thread, which then executes the message's `run` method. If the receiving thread were to execute the `run` method directly, then a deadlock could occur if the `run` method were to send a reply message back to the sender (a common occurrence). The RMI-based implementation incurs two costs over the raw RMI benchmark: executing the message's `run` method, and inter-thread communication. This cost difference is visible for small messages, but becomes insignificant when sending larger messages. The UDP-based implementation is slightly slower than the RMI-based implementation. It too incurs costs associated with executing the message's `run` method, and inter-thread communication. (Orig-

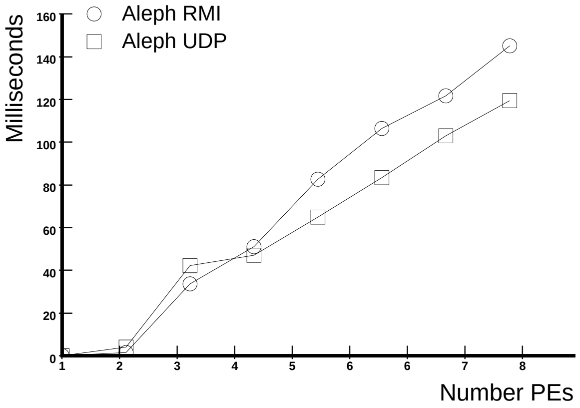

Figure 4: Null remote thread creation (Solaris)

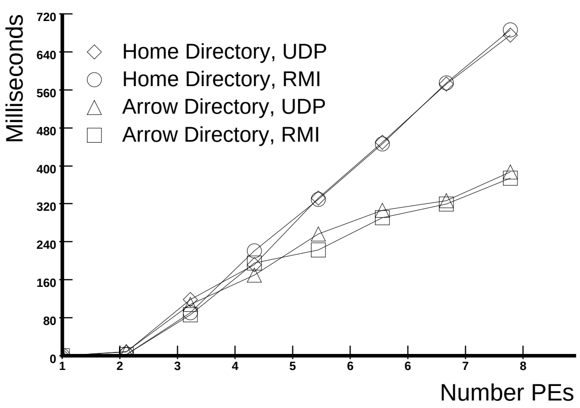

Figure 5: Incrementing shared counter (Solaris)

inally, the UDP-based implementation was faster, but the RMI-based implementation has been more thoroughly tuned).

The second benchmark measures how long it takes to create a null thread at a collection of PEs. This benchmark tests both point-to-point message-passing and event performance. The first PE creates a remote thread at each of the others, and, using a `Join` object, waits until they all finish. The Join object is implemented as an Aleph `Event` object, where the calling thread is the only listener. This interval encompasses the time needed to serialize the messages (sequentially) to transmit, deserialize, and run each message (in parallel), and to collect the responses via the Event object that implements the Join object. The results are reported Figure 4. To save space, we present only the Solaris results.

The third benchmark measures the time needed to increment a shared counter. The results are reported in Figure 5. Here, the "arrow" directory management protocol consistently outperforms the standard "home" directory structure. An algorithmic analysis of this performance difference lies beyond the scope of this abstract, but is available elsewhere [9].

5.1 Status

Aleph has been tested on Digital Unix, SUN Solaris, Linux, Windows NT and Windows 95. Installing Aleph on a new platform typically requires changing a few lines in a single configuration file (usually just the pathname of the java interpreter). The latest version of the Aleph toolkit is available via

`http://www.cs.brown.edu/~mph/aleph.html`.

The benchmarks described above can be found in the `bench` subdirectory.

6 Related Work

Pioneering work on DSM systems includes Ivy [16], Munin [5], Treadmarks [14], Midway [6], and others. Early work on language support for DSM includes Linda [2] and Orca [4]. The early Aleph design was substantially influenced by experience using the Cid DSM system [18]. In Cid, as in CRL [13], 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.

We are aware of two other DSM projects based on Java: Java/DSM [24], and Mocha [20]. Java/DSM implements a parallel Java Virtual Machine (JVM) running on top of Treadmarks [14]. The Infospheres [8] 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 [23] provides Java-based support for distributed systems with a focus on "federating" distinct services.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

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

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[4] 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.

[5] 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.

[6] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[7] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[8] 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.

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

[10] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, December 1997.

[11] Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994.

[12] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. Pvm 3 user'sguide and reference manu. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[13] 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.

[14] 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.

[15] B. N. Levine and J.J. Garcia-Luna-Aceves. A comparison of reliable multicast protocols. *Multimedia Systems Journal (ACM/Springer)*, 6(5), August 1998.

[16] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[17] Sun MicroSystems. Java remote method invocation specification. `www.javasoft.com-/products/jdk/1.1/download-pdf-ps.html`, 1997.

[18] 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.

[19] A. Pope. *The CORBA reference guide: Understanding the Common Object Request Broker*. Addison-Wesley, 1997.

[20] 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.

[21] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A framework for protocol composition in horus. In *Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing (PODC'95)*, pages 80–89, August 1995.

[22] 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.

[23] J. Waldo. Jini architecture overview. `www.javasoft.com/products/jini-/whitepapers/index.html`, 1998.

[24] W. M. Yu and A. L. Cox. Java/DSM: a Platform for Heterogeneous Computing. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.