REPLICATION TECHNIQUES FOR SPEEDING UP PARALLEL APPLICATIONS ON DISTRIBUTED SYSTEMS

Henri E. Bal *
M. Frans Kaashoek
Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands

Jack Jansen

Centrum voor Wiskunde en Informatica
Kruislaan 413
1098 SJ Amsterdam
The Netherlands

Email: bal@cs.vu.nl

*This research was supported in part by the Netherlands organization for scientific research (N.W.O.) under grant 125-30-10.

SUMMARY

Most methods for programming loosely-coupled systems are based on message-passing. Recently, however, methods have emerged based on ''virtually'' sharing data. These methods simplify distributed programming, but are hard to implement efficiently, as loosely-coupled systems do not contain physical shared memory. We introduce a new model, *the shared data-object model*, that eases the implementation of parallel applications on loosely-coupled systems, but can still be implemented efficiently.

In our model, shared data are encapsulated in passive data-objects, which are variables of user-defined abstract data types. To speed up access to shared data, data objects are replicated. This ability to replicate objects is a significant difference with other object-based models (e.g., Emerald and Amber). Also, by replicating logical objects rather than physical pages, our model has many advantages over shared virtual memory systems.

This paper discusses the design choices involved in replicating objects and their effect on performance. Important issues are: how to maintain consistency among different copies of an object; how to implement changes to objects; and which strategy for object replication to use. We have implemented several options to determine which ones are most efficient.

1. INTRODUCTION

Distributed systems are becoming increasingly popular for running large-grain parallel applications. These systems are easy to build and extend, and offer a good price/performance ratio. The issue of how to program parallel applications that use many loosely-coupled machines is still open. Traditional programming methods are based on some form of message-passing [1]. More recently, methods have emerged based on sharing data. Since distributed systems lack shared memory, this sharing of data is logical, not physical.

For many applications, support for shared data makes programming easier, since it allows processes on different machines to share state information. The main problem, however, is how to implement it efficiently on memory-disjunct architectures. In this paper we introduce a new model providing shared data and we discuss efficient implementation techniques for this model, based on *data replication*.

Several systems exist that use replication for implementing shared data. Probably the best known example is Kai Li's Shared Virtual Memory (SVM) [2]. This system gives the user the illusion of a shared memory. It stores multiple read-only copies of the same page on different processors. Each processor having a copy can read the page as if it were in normal local memory. Other systems providing replicated shared data are surveyed in [3, 4].

The model studied in this paper is called the *shared data-object model*. It is intended for implementing parallel applications on distributed systems. The unit of replication in our model is not dictated by the system (as in the SVM), but is determined by the programmer. Shared data are encapsulated in passive *data-objects**, which are variables of user-defined abstract data types. An abstract data type has two parts:

- A specification of the operations that can be applied to objects of this type.
- The implementation, consisting of declarations for the local variables of the object and code implementing the operations.

Instances (objects) of an abstract data type can be created dynamically, each encapsulating the variables defined in the implementation part. These objects can be shared among multiple processes, typically running on different machines. Each process can apply operations to

* We will sometimes use the term "object" as a shorthand notation for data-objects. Note, however, that unlike in most parallel object-based systems, objects in our model are purely passive.

the object, which are listed in the specification part of the abstract type. In this way, the object becomes a communication channel between the processes that share it.

The shared data-object model uses two important principles related to operations on objects:

1. All operations on a given object are executed *atomically* (i.e., *indivisibly*). To be more precise, the model guarantees *serializability* [5] of operation invocations: if two operations are executed simultaneously, then the result is as if one of them is executed before the other; the order of invocation, however, is nondeterministic.
2. All operations apply to *single* objects, so an operation invocation can modify at most one object. Making *sequences* of operations on different objects indivisible is the responsibility of the programmer.

These two principles make the model easy to understand and efficient. The first principle makes our model fundamentally different from Agora [6] and the problem-oriented shared memory [7], which do not have this consistency constraint. The second principle makes the model efficient to implement, since it avoids expensive atomic transactions on multiple objects stored on different processors.

In our experience thus far, the model provides sufficient support for many parallel applications. Distributed applications like banking and airline reservation systems can profit from more support (e.g., atomic multi-object operations), but such applications are not our major concern here. Also, parallel applications on *closely-coupled* (shared-memory) systems can use a finer grain of parallelism (e.g., parallelism within objects), but again these are not the type of applications we are interested in here. These issues are addressed by other models, such as atomic transactions and concurrent object-oriented programming and are not the topic of this paper.

Our model also differs from the *object-based* models supported by Emerald [8] and Amber [9]. Objects in these languages are migrated between processors, but are not replicated. An Emerald object, for example, can be active—it may contain a process—and should not be replicated. As another important difference, our model completely hides the distribution of objects and lets the implementation determine where to store (and replicate) objects. Emerald and Amber ultimately rely on the programmer to specify the most efficient location for an object.

We have designed a new programming language called *Orca*, based on this model. Orca is intended for implementing distributed *user* applications. In particular, the language is intended for parallel, high-performance applications. Orca is *not* an object-oriented language. Rather, it is a simple, procedural, type-secure language. It supports abstract data types, processes, a variety of data structures, modules, and generics.

Various implementations of Orca on different hardware configurations have been in use for three years. The language, its implementation, and use are described elsewhere [10, 11, 12].

In the rest of this paper we will study replication techniques for the shared data-object model. In Section 2, we will describe the space of possible design choices. The most important issues are (1) updating versus invalidation of copies, (2) the protocols used for updating or invalidating copies, and (3) the degree of replication. As we will see, the best choice depends on the communication primitives supported by the underlying distributed system. We will study two important cases. In Section 3, we will look at an implementation of the model using point-to-point message passing. In Section 4, we will discuss a second implementation, based on reliable multicast messages. The two implementations cover a broad

spectrum of design choices. In Section 5, we will measure the performance of the two implementations on distributed hardware. In Section 6, we will present our conclusions and compare our work with that of others.

2. DESIGN SPACE

The technique of data replication in distributed systems is typically used to increase the availability and reliability of the data in the presence of processor failures and network partitions [13, 14, 15, 16, 17]. For example, if multiple copies of the same logical data are stored on different processors, the data can still be accessed if some of the processors are down.

In contrast, we use replication primarily for speeding up access to shared data and for decreasing the communication overhead involved in sharing data. The general idea is to replicate an object on those processors that frequently access it. A copy may be accessed by all processes running on the same processor, without sending any messages, as shown in Figure 1.

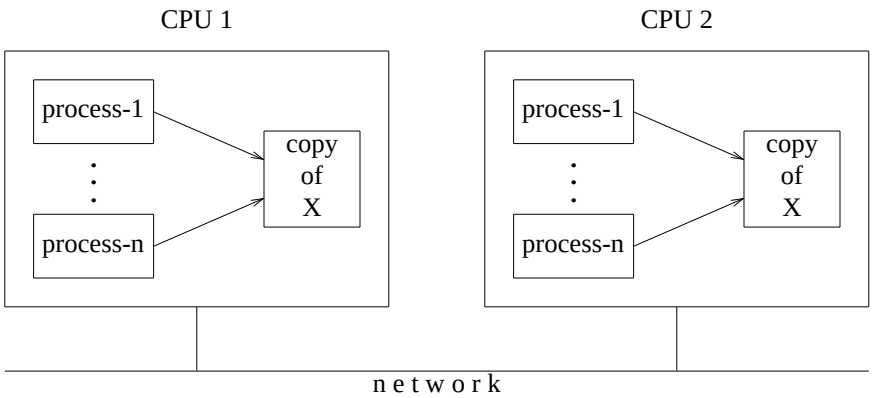

Figure 1. Replication of data objects in a distributed system. Each processor contains multiple processes running in pseudo-parallel. These processes belong to a single job and run in a single address space, so they can share copies of objects.

It is useful to distinguish between *read* operations and *write* operations on replicated data: a read operation does not modify the data, while a write operation (potentially) does [15]. For our model, we define a read operation as an operation that does not change the internal data of the object it is applied to.

The primary goal of replicating shared data-objects is to apply read operations to a local copy of the object, if available, without doing any interprocess communication. On a write operation, all copies of the object except the one just modified must be invalidated or updated. To deal with this problem, communication will be needed, so write operations involve communication.

This scheme is a departure from techniques that replicate for availability. These techniques in general need interprocess communication for every read and write operation. With our approach, read operations are executed locally. Since, for many parallel applications, read operations far outnumber write operations [18], this is a significant advantage.

The second goal of replication is to increase parallelism. If an object is stored on only one processor, each operation must be executed by that processor. This processor may easily become a sequential bottleneck. With replicated objects, on the other hand, all processors can simultaneously read their own copies. Since a read operation does not change its object, it can be executed concurrently with other read operations without violating the serializability principle.

The effectiveness of replication depends on many factors. One important factor is the ratio of read and write operations on objects, which is determined by the user application. Another factor is the overhead in execution time for reading or writing objects. These costs are determined by the implementation of the model. They depend on:

- The action undertaken after each write. If each write operation *invalidates* all copies, a subsequent read operation will need to do communication. If, on the other hand, all copies are *updated*, this disadvantage disappears, but write operations will become more expensive
- The protocol used for invalidating or updating copies. Many protocols exist (e.g., owner protocols, two-phase update protocols), each with their own advantages and disadvantages.
- The replication strategy. If an object is replicated everywhere, each read operation can be applied to a local copy, which is much cheaper than doing the operation remotely. On the other hand, writing an object that has many copies will be more expensive than writing a non-replicated object.

In the following subsections we will study these design choices in more detail.

2.1. Invalidation versus Updating of Copies

If a write operation is applied to a replicated object, its copies will no longer be up-to-date. There are two different approaches for dealing with this problem. The first scheme is to *invalidate* all-but-one copies of the object. The second scheme is to *update* all copies in a consistent way.

With invalidation (or *write-once*), each object is initially stored on only one processor, say *P*. If another processor wants to do a read operation on the object, it fetches a copy of the object from *P*. In this way, the object automatically gets replicated. On a write operation, all-but-one copies are thrown away.

The alternative scheme is to update (or *write-through*) all copies of an object after each write operation. A problem here is how to update all copies *in a consistent way*. The shared data-object model guarantees that all operations on objects are executed indivisibly. Hence, updating of all copies should appear as one indivisible action. On systems supporting only point-to-point communication, this is hard to do. In essence, a *2-phase* protocol is needed, as we will see. If reliable indivisible multicast messages are available, updates become much simpler, as we will discuss in Section 4.

There are several important differences between invalidation and update schemes. For one thing, keeping copies up-to-date is more complicated than invalidating copies, so the update scheme may require more messages to implement a write operation. Also, update messages will be larger than invalidation messages. An invalidation message merely needs to specify the object to be invalidated. An update message will either contain the new value of the object or the parameters of the write operation, whichever is more efficient.

On the other hand, the update scheme also has several advantages. If an object is read after it has been written, the invalidation scheme will have to fetch the current value of the object from a remote processor. With the update scheme, this value will still be stored locally, so no messages need be sent at all.

In conclusion, which of the two schemes is most efficient depends on:

1. The costs of the update protocol.
2. The size of the object.
3. The size of the parameters of the write operation.

4. Whether the write operation is followed by a read operation or by another write operation.

Kai Li argues that, for the Shared Virtual Memory system, an update scheme is inappropriate [19]. In addition to being almost impossible to implement, it will cause a page fault on *every* write instruction. In our model, however, this disadvantage is far less severe. Users can define write operations of any complexity on shared objects. As replicas are updated after each operation—rather than each machine instruction—updating will be less expensive than in the SVM. In addition, the SVM would require a whole page to be transmitted after every write. With our approach, shared objects frequently are much smaller than a page; furthermore, large objects can usually be updated efficiently by transmitting the operation and its parameters, instead of the new value of the object.

2.2. Invalidation and Update Protocols

The protocol used for invalidating or updating copies of objects must make sure that simultaneous operations on the same object are executed indivisibly. The simplest way to implement this is to serialize all write operations (i.e., to execute them one at a time, in a mutually exclusive way). This is the approach taken by all our implementations.

In an invalidation scheme, mutual exclusion can be achieved by selecting one copy of each object as the *primary copy*. In the simplest scheme, all write operations are directed to the processor containing the primary copy. On receiving a write operation, the processor first invalidates all secondary copies and then applies the operation to the primary copy. When a processor executes a read operation, it locates the primary copy and asks for the value of the object. A more sophisticated scheme allows the primary copy to move from one processor to another. Kai Li compares several of these schemes and analyzes their performance [19].

In an update scheme, mutual exclusion can be achieved in at least two ways. One way is to appoint one copy of each object as *primary copy* and direct all write operations to the processor containing the primary copy. This node will execute the write operations one by one and propagate their effects to all other copies, called *secondary copies*. An alternative approach is to treat all copies as equals and use a *distributed protocol* that takes care of mutual exclusion. With such a protocol, each processor can initiate a write operation on an object. Coordination is needed to prevent interference of simultaneous write operations on the same object.

2.3. Replication Strategies

Replicating a shared data-object is only useful if it is read relatively often. Thus, simply replicating all objects on all processors is unlikely to be efficient. In general, we can distinguish between several *strategies* for replication:

No replication:	Each object is stored on one specific processor.
Full replication:	Each object is replicated on all processors.
Partial replication:	Each object is replicated on some of the processors, based on (a) compile-time information, (b) run-time information, or (c) a combination of both.

The first approach is used in most parallel object-based languages. In this case, all operations on a given object are executed by the same processor. For many applications, this may easily lead to sequential bottlenecks and high communication overhead.

The second approach indiscriminately replicates all shared objects on all processors. It will be most effective for architectures supporting fast reliable multicast messages, since these will allow efficient updating or invalidation of all copies.

The third strategy selectively replicates objects, based on information gathered by either the compiler, the run time system (RTS), or both. With this approach, several scenarios are possible. For example, the compiler may disable replication of objects that do not have any read operations at all. Also, if a processor does not contain any processes that share a given object, it is unnecessary to store a copy of the object on that processor.

The most advanced scheme based on partial replication is to let the RTS decide dynamically where to replicate each object. For example, the RTS may keep track of read and write operations on an object issued by each processor, to determine which processors frequently read the object. If the read/write ratio exceeds a certain threshold, a replica of the object is created dynamically on that processor. This strategy is most suitable if communication is slow, so the overhead of maintaining statistics is worthwhile.

2.4. Discussion

We have discussed several design choices related to replication of objects. In general, it is hard to determine which ones will give the best overall performance. Furthermore, different types of distributed systems may require different design decisions. In particular, the communication primitives provided by the system are very important.

In the next two sections we will examine two existing implementations of the shared data-object model. Each implementation is a run time system for Orca. Both RTSs use the same hardware: a collection of 10 MC68030 CPUs connected by a 10 Mbit/sec Ethernet®, but use different communication primitives and consistency protocols.

The first RTS uses a rather conventional software organization, based on point-to-point message passing and 2-phase update protocols. Its novelty is its dynamic replication strategy based on run-time statistics. Although statistics are used frequently in distributed data bases, they are uncommon in distributed programming languages (the only exception we know of is [20]).

The second RTS is based on a novel multicast protocol [21]. This protocol provides the necessary semantics for keeping all copies of each object consistent. Also, it is optimized for parallel applications, in which processes communicate fairly often.

Both run time systems are implemented on top of the Amoeba distributed operating system [22] and use the FLIP routing protocol [23], which supports point-to-point communication as well as multicast.

3. AN IMPLEMENTATION USING POINT-TO-POINT COMMUNICATION

The first run time system we describe uses only point-to-point messages (Amoeba Remote Procedure Call) for interprocess communication. Below, we will look at each of the three design issues discussed in Section 2 and motivate our choices. In Section 5.1 we will describe the performance of this system.

3.1. Invalidation versus Updating

The first issue is the choice between an invalidation or an update scheme. With point-to-point messages it is expensive to update all copies of an object in a consistent way. Simultaneous write operations on the same object can be serialized using a primary copy protocol, as described in Section 2.2. A harder problem is how to achieve serializability if a sequence of operations on *different* objects is executed.

Suppose a program uses two objects, X and Y, that have their primary copies on different processors. If X and Y are written simultaneously, either all processors should observe the change to X first or all processors should observe the change to Y first. Under no circumstances should these two events be mixed, since that would violate serializability. Because of this restriction, it does *not* suffice to implement a write operation by sending it to the primary-copy site and having this site forward it to the secondary-copy sites [12].

The problem can be solved using a more complicated and expensive update protocol [12]. Therefore, updating copies will be expensive, so it is not clear whether updating will be more efficient than invalidation. We have decided to implement both options and to determine experimentally which of the two is best.

3.2. The Protocols

In this section we will discuss the protocols for invalidating or updating copies, using point-to-point messages.

The Invalidation Protocol

The invalidation protocol is quite simple. A process that wants to invoke a write operation on a shared object sends the operation and the parameters to the processor containing the primary copy of the object. This processor locks the object and sends point-to-point *invalidate* messages to all processors containing a secondary copy. If a secondary-copy site receives this message, it throws away its local copy of the object and sends back an acknowledgement. As soon as the primary-copy site has received all acknowledgements, it updates and unlocks the primary copy. This protocol requires two messages for each secondary copy. In addition, if the primary copy is not on the invoker's processor, two more messages are needed for updating the primary copy.

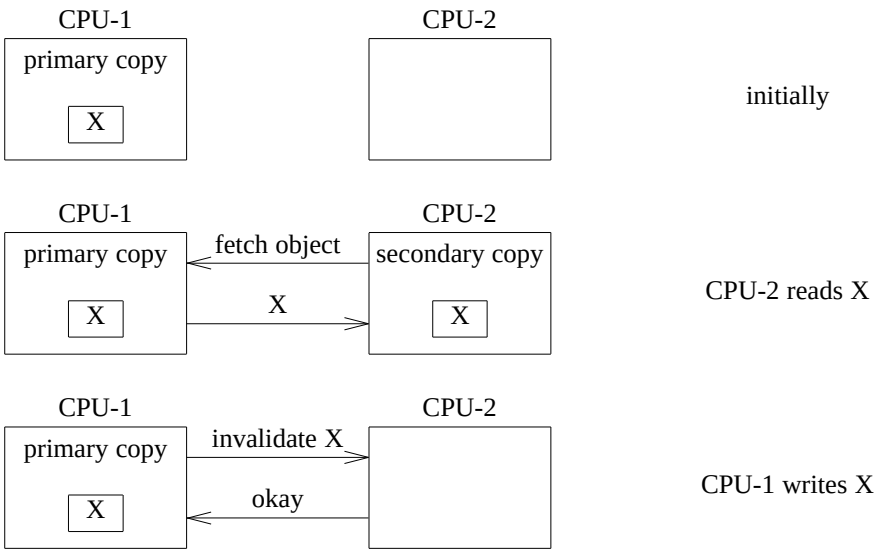

Figure 2. The invalidation protocol. Initially, only CPU-1 contains a copy of X, the primary copy. When CPU-2 wants to read X, it sends a *fetch-object* message to the primary-copy site to obtain a (secondary) copy. Upon a write operation, the secondary copies are deleted.

If a process *P* wants to do a read operation on an object of which it does no longer have a local copy, it sends a *fetch-object* message to the primary-copy site (see Figure 2). If this processor has not yet received all acknowledgements, the primary copy will be locked and *P* will temporarily be blocked. When the object is unlocked, the primary-copy site sends a new copy to *P*.

The Update Protocol

Updating all copies of an object in a consistent way is more difficult than invalidating them. The real problem is to guarantee serializability, as discussed above. We solve this problem using a *2-phase* primary copy update protocol. The protocol updates copies by sending the operation and its parameters to the secondary-copy sites. For most programs, this is more efficient than transmitting the new value of the object.

During the first phase, the primary copy of the object is locked and a *lock-and-update* message is sent to all secondary-copy sites. This message specifies an object, an operation to be applied to the object, and the parameters of the operation. When a site receives the *lock-and-update*, it locks the local copy of the object and applies the operation to it. Next, it sends an acknowledgement to the primary-copy site, while still keeping its local copy locked. In the mean time, the primary-copy site waits for all acknowledgements and then sends an *unlock* message to all sites. The *unlock* message causes all copies of the object to be unlocked.

The 2-phase update protocol guarantees that no process uses the new value of an object while other processes are still using the old value. The new value is not used until the second phase. When the second phase begins, all copies contain the new value. Simultaneous write-operations on the same object are serialized by locking the primary copy. The next write-operation may start before all secondary copies are unlocked. New requests to *lock-and-update* a secondary copy are not serviced until the *unlock* message generated by the previous write has been handled.

This protocol requires three reliable messages for each secondary copy. In the first phase, one request to *lock-and-update* the object is sent plus an acknowledgement for this request. In the second phase, an *unlock* message is sent. To update an object whose primary copy is located on a remote processor, two extra reliable messages are needed. Since our implementation uses Amoeba RPC rather than 1-way asynchronous messages, there is also some overhead in sending reply messages for the RPCs. The implementation is optimized, however, to overlap regular computations with sending reply messages, so the latter overhead is small.

The usage of a 2-phase update protocol in a language RTS is certainly not new. Languages based on atomic transaction (e.g., Argus [24]) also use 2-phase protocols. In our model, however, a 2-phase protocol is used for updating copies of the *same* object, rather than for updating many different objects. Our implementation does not have to deal with the case that part of the objects are locked or that part of the operations fail. In particular, our RTS does not have to maintain multiple versions of objects. Therefore, our implementation is much simpler than that of transaction systems.

3.3. Replication Strategy

With the above protocols, the costs of invalidating or updating *N* copies of an object will grow linearly with *N*. As a result, it will be expensive to replicate all objects on all processors. Our implementation therefore uses a partial replication strategy, based on run time statistics. Although this incurs some overhead on operations, communication costs can be reduced significantly. As communication in distributed systems still is expensive (on the order of milliseconds), this approach is attractive.

Initially the system contains one copy for each object: the primary copy. If some processor frequently tries to read the primary, a secondary copy will be created, so that future read operations can be applied to the local copy without sending any messages. Write operations are always directed to the primary copy.

In the invalidation scheme the owner of the primary copy invalidates all secondary copies before performing the write operation. A subsequent read operation on the same object always has to go to the processor containing the primary copy. So, the number of secondary copies of a given object is determined by its read/write pattern.

In the update scheme, the processor containing the primary copy of an object keeps track of the number of remote read and write operations issued by each processor. The overhead of maintaining these statistics is negligible compared to the total costs of remote operations. As soon as the read/write ratio of a remote processor exceeds a certain threshold, the RTS creates a copy of the object on that processor.

Each processor having a secondary copy keeps track of the ratio of local read operations and (global) write operations. If the overhead in updating the copy exceeds the time saved in doing read operations locally, the RTS discards the local copy. From then on, all operations on the object will be done remotely.

With both the invalidation and update protocol, all write operations are forwarded to the processor containing the primary copy of the object. If the RTS discovers that an object is written frequently by a machine different from the one containing the primary copy, the RTS may decide to *migrate* the primary copy to that machine. Again, statistics are used to determine to best location for an object. If an object is migrated, precautions are taken for dealing with machines that are unaware of the object's new location.

4. AN IMPLEMENTATION USING MULTICAST COMMUNICATION

The second RTS uses Amoeba's indivisible reliable multicast protocol described in [21]. This protocol is highly efficient and usually only requires two packets (one point-to-point and one multicast) per reliable multicast. Sending a short message reliably to 10 processors, for example, takes 2.7 msec on the hardware described above.

In a distributed system supporting only point-to-point messages, serializability is difficult to achieve, because messages sent to different destinations may arrive with arbitrary delays. Some distributed systems (e.g., Ethernet-based systems) provide hardware support for sending a single message to multiple destinations simultaneously. More precisely, we are interested in systems supporting *indivisible reliable multicasts*, which have the following properties:

- A message is sent reliably from one source to a set of destinations.
- If two processors simultaneously multicast two messages (say m_1 and m_2), then either all destinations first receive m_1, or they all receive m_2 first, but not a mixture with some receiving m_1 first and others receiving m_2 first.

With this multicast facility, it becomes much easier to implement a protocol for consistent updating of all copies of an object. Basically, if a process wants to invoke a write operation on a shared object, it multicasts the operation to all processors. Since all processors receive all messages in the same order, all operations on shared objects are executed in the same order everywhere.

We have implemented an indivisible reliable multicast protocol in software on top of Ethernet. The basic idea behind the protocol is that one of the nodes be designated as the *sequencer*. If a node wants to multicast a message, it first sends this message to the sequencer, using point-to-point communication. The sequencer assigns the message the next global sequence number and then multicasts the message and its sequence number. When a node receives such a multicast message, it checks the sequence number to see if it has missed any multicasts. If so, it requests the sequencer to provide it with the missing message (the sequencer stores these in order to provide this recovery service).

The above protocol sends each message over the network twice. For large messages it is more efficient to let the sender broadcast the message itself, and have the sequencer broadcast a (small) acknowledgement message containing the sequence number. The protocol therefore uses the first approach for small messages and the second approach for large messages. For all examples discussed in this paper, the first method is used.

Of course, there are many more issues involved in the protocol, such as buffer management of messages, group management, and crashes of the sequencer or regular nodes. These issues are described in [21].

With the protocol outlined above, programs need not worry about lost messages. Recovery of communication failures is handled automatically and transparently by the protocol. Efficiency is obtained by optimizing the protocol for no communication failures, as these rarely happen with current state of microprocessor and network technology.

4.1. Invalidation versus Updating

Reliable multicasting is useful for invalidation as well as updating. In both cases, a single reliable multicast message is needed for a write operation. If an object is written very frequently and hardly ever read, the invalidation scheme will be more efficient, since fewer messages are needed and invalidation messages are shorter than update messages.

In general, however, the update scheme will be more efficient. Suppose, for example, that every processor reads a given object exactly once after it has been written. With P processors, the invalidation scheme requires a single (short) reliable multicast message for invalidating the copies and $2P$ point-to-point messages for fetching the object (or doing the read operation remotely). As a reliable multicast usually costs two physical messages, in total there are $2P + 2$ messages. In contrast, the update scheme requires only one reliable multicast message. So, even in the case that each write is followed by only a single read operation, the updating performs better than invalidation.

With the reliable multicast protocol we use, a multicast message is hardly more expensive than an RPC. So, unless the read/write ratio of operations is close to zero, the update scheme will have a better performance. We have therefore only implemented the update scheme.

4.2. The Update Protocol

As in the RPC run time system, indivisibility of write operations is obtained by executing them in a mutually exclusive way. With indivisible multicast, mutual exclusion comes for free. The communication primitive imposes a single system-wide global ordering on all write operations. Unlike the point-to-point scheme, there is no risk of different processors updating their copies in an inconsistent way. Also, there is no need to distinguish between primary and secondary copies of an object.

The distributed update protocol we use works as follows. Each processor maintains a queue of messages that have arrived on the processor but that have not yet been handled. As all processors receive all messages in the same order, the queues on all processors are basically the same, except that some processors may be ahead of others in handling the messages at the head of the queue.

If a process wants to execute a write operation on a shared object X, it multicasts an *update* message to all processors (including its own processor) and then blocks. The message contains the name of the object, the operation, and its parameters. The update message will be appended to the tail of each queue.

Each processor handles incoming messages in its queue in strict FIFO order. A

message may be handled as soon as it appears at the head of the queue. To handle an *update* message, the message is removed from the queue, the local copy of *X* is locked, the operation is applied to the local copy, and finally the local copy is unlocked. If the message was sent by a process on the same processor, that process is made active again.

The protocol described above correctly implements the serializability requirement. The protocol guarantees that all processors observe changes to shared objects *in the same order*. Note that it does not provide a total (*temporal*) ordering [25] among operations. Suppose Processor P1 initiates a write operation on object *X* and, a few microseconds later, Processor P2 reads the value of *X*. The *update* message for *X* sent by P1 need not have even reached P2 yet, so P2 may still use the old value of *X*. This scenario is in accordance with the semantics of our model, however, which merely requires serializability of operations.

4.3. Replication Strategy

The multicast RTS replicates all objects on all processors. In other words, it uses the full replication strategy. This strategy was chosen, because it simplifies the implementation. The RTS does not have to keep track of which object is used by which processor.

In some cases, full replication may be less efficient than partial replication. Suppose, for example, process P1 wants to send information to another process P2 through an object shared between them. As the object will be replicated everywhere, all processors in the system will receive P1's update message, even though only P2 is really interested in it.

The overhead of sending the message everywhere usually is not dramatic, however. With our reliable multicast protocol, the elapsed time for a multicast message hardly depends on the number of destinations. The main disadvantage of full replication then is the fact that each processor will be interrupted once for each write operation. With partial replication, this CPU overhead would be less.

5. PERFORMANCE

There are several ways to measure the performances of the replication techniques. The approach taken in [12] is to implement several user applications in Orca, execute them on the different run time systems, and measure the speedups. Applications we have looked at are matrix multiplication, the all-pairs shortest paths problem, branch-and-bound, alpha-beta search, and successive overrelaxation.

In this paper we will first look at the basic times for reading, writing, updating, and invalidating shared objects. In this way we can determine under which circumstances a technique is most effective. Next, we will look at the access patterns used by real programs and use the basic times for determining the efficiency of the different strategies for these applications.

To determine the performance improvements due to replication, we have performed two experiments. In the first experiment, we have measured the costs of incrementing a replicated 4 byte integer object as a function of the number of replicas. In the second experiment we measured the cost for updating an entire 1 Kb array object. These two types of objects occur frequently in application programs [12].

5.1. Performance of the RPC Run Time System

Figure 3 shows the basic execution times for the run time system that uses Amoeba RPC. The figure shows the costs for invalidating *N* copies of an object and for updating 4-byte and 1 Kb objects. Invalidating a copy involves sending a short message containing an object identifier, so the invalidation costs do not depend on the size of the object.

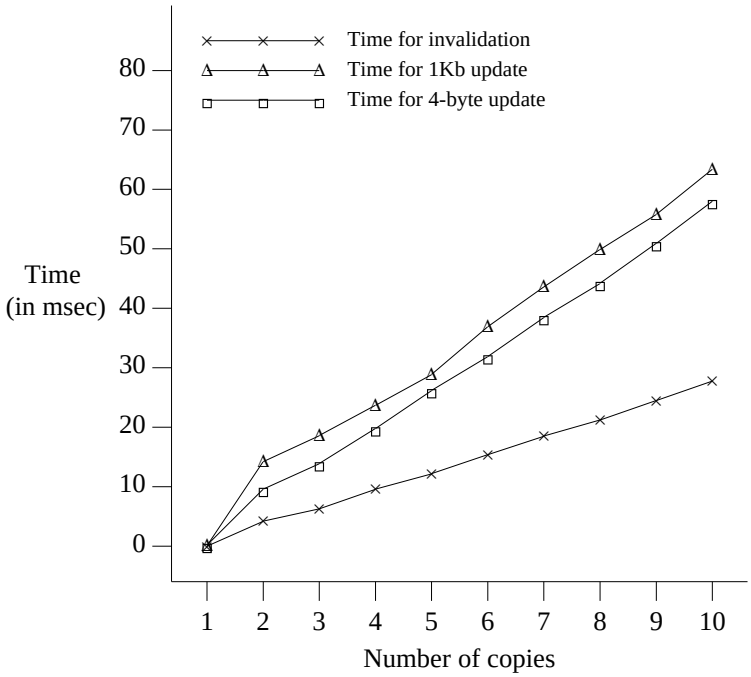

Figure 3. Time for updating replicated objects using invalidation and 2-phase update protocol.

We have also measured the costs for doing a read or write operation on a remote object. For a 4-byte object, remote reads cost 5.3 msec and remote writes costs 4.2 msec; for a 1Kb object, the costs are 12.7 msec and 7.6 respectively. The reason why writes are cheaper than reads is that, due to inefficiencies in the current compiler, read operations copy their data more often.

As expected, the costs to update or invalidate copies after a write operation grow linearly with the number of copies. Therefore, selective replication is worth while.

For a small (4-byte) object, updating 10 copies costs 58.0 msec. Invalidating 10 copies takes 27.8 msec; in addition, re-installing copies costs 5.3 msec per copy (i.e., the costs of a remote read operation). If the object is read by 6 or more processors immediately after it has been written, updating will outperform invalidation, since 27.8 + 6*5.3 > 58.0.

In contrast, if a small object is written twice without being read, the invalidation scheme is more efficient. In this case, the update costs are 2*58.0 = 116.0 msec. Invalidating and re-installing all 9 secondary copies takes 27.8 + 9*5.3 = 75.5 msec.

For large (1 Kb) objects, updating 10 copies takes 63.4 msec. If 3 processors re-install the object after a write operation, the invalidation scheme costs 27.8 + 3*12.7 = 65.9, which is slower than updating. If fewer than 3 processors read the object, invalidation is more efficient.

If a large object is written twice and then read by all processors, updating all copies costs 2*63.4 = 126.8 msec; invalidating the copies once and re-installing them takes 27.8 + 9*12.7 = 142.1 msec, so updating is still cheaper. If a large object is written three or more times successively, invalidation will be more efficient.

A case that occurs frequently in user programs is a large object that is written through an operation with only a few bytes of parameters (e.g., a 1Kb array of which only 1 element is changed). In this case, updating will often be more effective. For example, updating 10 such copies will take about 58.0 msec, while invalidating and re-installing the entire array will cost 142.1 msec. Even if the object is changed twice before being read, updating is

significantly more efficient.

The performance measurements also show that the partial replication scheme is more efficient than a scheme that does not replicate objects, if the object is read relatively frequently. If, for example, a given 4-byte object is not replicated, each (remote) read operation will take 5.3 msec, so partial replication clearly pays off.

5.2. Performance of the Multicast Run Time System

The cost for updating replicated objects using the distributed update protocol described in Section 4 are depicted in Figure 4. As can be seen, the costs are almost independent of the number of replicas. This is what we would expect, since in our multicast protocol sending a reliable multicast message costs only two physical messages, independent of the number of receivers. The only overhead is sending one *state message* after a processor has received a certain number of messages. (This state message is only required if a processor does not multicast messages itself; if it does multicast messages, the state message is piggybacked.)

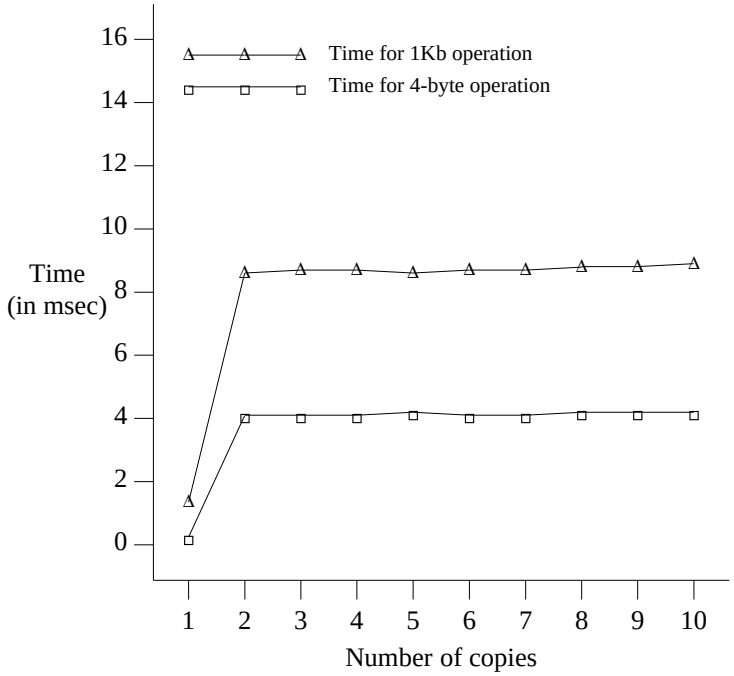

Figure 4. Time for updating replicated objects using the distributed update protocol.

For both 4 byte and 1 Kb objects, the update costs are lower than or equal to the costs for remote read and write operations in the RPC run time system (see Section 5.1). The reason is that the RPC system has a higher overhead. In particular, it does more context switching, because it uses auxiliary threads for queuing messages.

Since updates are cheap, replication usually reduces the communication costs. One exception is an object with a low read/write ratio. In this case, the overhead of updating the replicas after each write will invalidate the gains of replication. The second exception is an object that is hardly ever accessed by remote processors. In both cases, it would be better not to replicate the object at all. In conclusion, a good strategy for small objects would be to either replicate a given object everywhere or not at all.

5.3. Access patterns of example applications

Which replication strategy is most efficient for a given application depends not only on the basic performance figures presented above, but also on the access patterns of the application. In this section, we will look at how real Orca applications read and write shared objects. Using the measurements given above, we can determine the effectiveness of each strategy. The applications we will look at are the traveling salesman problem and the all-pairs shortest paths problem. In our examples, we will assume that we use 10 CPUs.

The Traveling Salesman Problem

In the Traveling Salesman Problem (TSP) it is required to find the shortest route for a salesman to visit each city in a given set exactly once. The problem is solved in Orca using a master/slave type of program based on a branch-and-bound algorithm. The master generates partial routes and stores them in a job queue. Each slave repeatedly takes a job (route) from the queue and generates all possible full paths starting with the initial route. All slaves keep track of the current shortest full route. As soon as a slave finds a better route, it gives the length of the route to all the other slaves. This value is used to prune part of the search tree.

This application uses two shared objects that are important to our discussion. First, the master and slaves share a *job queue* object. All operations applied to this object are write-operations, since both adding a job to the queue and deleting a job from the queue modify the queue's data structures. Hence, the best strategy is not to replicate this object at all. The RPC RTS will store the object only on the master processor; the slave processors will access the object doing remote write operations.

As a typical example, consider a TSP problem with 12 cities, where the master generates initial routes containing 2 cities. So, the master generates 11*10 = 110 jobs. Each job description is only a few bytes. The multicast RTS will thus do 110 multicasts to 10 CPUs, which takes 110*4.2 = 462 msec. The RPC RTS (both the updating and invalidating version) will do 110 remote write operations, also taking 110*4.2 = 462 msec. So, the communication costs are the same in both systems, but the multicast system has the disadvantage of generating more interrupts for updating copies.

TSP uses another object (the bound) for keeping track of the current best solution. This object is shared among all slave processes. Measurements of the program for a 12-city problem show that this object may be read a million times and updated only a few times [12]. After the object has been changed (i.e., a slave has found a better route for the salesman), this new value is read many times by all the slaves. Thus the best strategy is to replicate the variable everywhere and update all copies whenever the variable changes.

Here, the multicast RTS has a performance advantage. If the object is updated ten times, this RTS will multicast ten write operations, which takes only 10*4.2 = 42 msec. The RPC RTS using the update protocol will take 10*58.0 = 580 msec. The invalidating RTS will invalidate the object ten times and then re-install it everywhere, taking 10*(27.8 + 9*5.3) = 755 msec. The total execution time of the TSP program on 10 CPUs is about 90 seconds, so the impact of this communication overhead is relatively small. Still, the multicast RTS achieves slightly better speedups. The second problem we will discuss has a much higher communication overhead.

The All-Pairs Shortest Paths Problem

In the All-pairs Shortest Paths (ASP) problem it is desired to find the length of the shortest path from any node i to any other node j in a given graph with N nodes. The parallel algorithm we use is similar to the one given in [26], which is a parallel version of Floyd's algorithm.

The distances between the nodes are represented in a matrix. Each processor contains a *worker* process that computes part of the result matrix. The parallel algorithm performs *N* iterations. Before each iteration, one of the workers sends a *pivot row* of the matrix to all the other workers. Since the pivot row contains *N* integers and is needed by all processors, this requires a nontrivial amount of communication.

The workers share an object containing all the pivot rows used for different iterations. Initially, this object is empty; after the final iteration, it will contain all the pivot rows used during the computation. The multicast RTS will replicate this object everywhere. If N=256, all copies of the object will be updated 256 times, each update operation taking a row of 256 integers (i.e., 1 Kb) as parameters. This will take 256*8.9 = 2278.4 msec.

The RPC RTS using the update protocol will likewise have 256*63.4 = 16230.4 msec of communication overhead. The total execution time of the program is on the order of 90 seconds, so this is a significant overhead. The invalidating RTS performs much worse, however. Re-installing the shared object would be prohibitively expensive, since the object may ultimately contain 256*256 integers, or 256 Kb data. We have measured that such an operation would cost about 16400 msec. On the other hand, letting each worker process obtain the pivot row from one processor also is very inefficient, since it requires 9 remote read operations of 1 Kb, taking 9*12.7 = 114.3 msec per iteration, or 114.3*256 = 29260.8 msec in total, which is almost twice as bad as for the update protocol. In conclusion, the invalidation protocol is not appropriate for ASP. With the update protocol, it is possible to obtain reasonable speedups [12], although far from linear. To obtain good (close to linear) speedups, the multicast protocol is required.

6. CONCLUSIONS

The model discussed in this paper allows programmers to define operations of arbitrary complexity on shared data-objects. In a loosely-coupled system, the model is implemented by replicating objects in the local memories of the processors. This ability to replicate objects is a significant difference with other object-based models, such as Emerald [8] and Amber [9]. We have studied several protocols for keeping all these copies consistent and we have looked at replication strategies.

We have described two implementations of the model. One implementation replicates objects everywhere and updates copies through a fast multicast protocol. The other implementation uses only point-to-point messages. In this case, partial replication and migration may be useful.

Which protocol or strategy for replication is most efficient depends on many factors, such as the costs of the update protocols, the size of the object and the parameters of the operations, and the read/write pattern of the application. In the future we intend to do a more detailed analysis of our protocols and strategies, using a large set of user applications. Also, we will look at the differences and resemblances between protocols for replication and coherence protocols for CPU caches [18, 27], non-uniform memory access (NUMA) architectures [28, 29], file caches [30, 31, 32], and distributed database systems [14]. Based on this analysis, we will try to improve our implementations.

Our model has several advantages over other models based on logically shared data. It provides a higher level of abstraction and, in many cases, is more efficient. Below, we will compare our model with several related ones.

Some systems provide the programmer a shared address space without guaranteeing coherency or consistency. In Agora [6] and the problem-oriented shared memory [7], for example, read operations can return *stale* data. Therefore, these systems do not make replication transparent to the user. Although these relaxations of the semantics make the systems

more efficient to implement, we feel that they do not provide a sound basis for a general-purpose parallel programming language. We prefer to have simple and easy-to-use semantics and therefore support consistency of replicated shared data.

Kai Li's Shared Virtual Memory supports memory coherency, but it has other disadvantages [33]. For example, it can only invalidate but not update copies of data. Also, the SVM will perform very poorly if processes on many different processors repeatedly write on the same page. This situation arises if multiple processors write the same variable, or if they write different variables placed on the same page.

Linda's Tuple Space [34] is another model that hides replication from the programmer. It provides a fixed number of low-level operations on shared data (tuples) [35]. Logical operations on shared data structures frequently consist of several low-level operations, each of which can require physical communication. In our model, the programmer can define a single high-level operation that does the job with lower communication costs.

7. REFERENCES

1. H.E. Bal, J.G. Steiner, and A.S. Tanenbaum, ''Programming Languages for Distributed Computing Systems,'' *ACM Computing Surveys* **21**(3), pp. 261-322 (Sept. 1989).
2. K. Li, ''IVY: A Shared Virtual Memory System for Parallel Computing,'' *Proceedings 1988 International Conference Parallel Processing (Vol. II)*, St. Charles, Ill., pp. 94-101 (Aug. 1988).
3. M. Stumm and S. Zhou, ''Algorithms Implementing Distributed Shared Memory,'' *IEEE Computer* **23**(5), pp. 54-64 (May 1990).
4. B. Nitzberg and V. Lo, ''Distributed Shared Memory: a Survey of Issues and Algorithms,'' *IEEE Computer* **24**(8), pp. 52-60 (Aug. 1991).
5. K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, ''The Notions of Consistency and Predicate Locks in a Database System,'' *Communications of the ACM* **19**(11), pp. 624-633 (Nov. 1976).
6. R. Bisiani and A. Forin, ''Multilanguage Parallel Programming of Heterogeneous Machines,'' *IEEE Transactions on Computers* **37**(8), pp. 930-945 (Aug. 1988).
7. D.R. Cheriton, ''Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems,'' *ACM Operating Systems Review* **19**(4), pp. 26-33 (Oct. 1985).
8. E. Jul, H. Levy, N. Hutchinson, and A. Black, ''Fine-Grained Mobility in the Emerald System,'' *ACM Transactions on Computer Systems* **6**(1), pp. 109-133 (Feb. 1988).
9. J.S. Chase, F.G. Amador, E.D. Lazowska, H.M. Levy, and R.J. Littlefield, ''The Amber System: Parallel Programming on a Network of Multiprocessors,'' *Proceedings of the 12th ACM Symposium on Operating System Principles*, Litchfield Park, AZ, pp. 147-158 (Dec. 1989).
10. H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, ''Orca: A Language for Parallel Programming of Distributed Systems,'' *IEEE Transactions on Software Engineering* (1992, to appear).
11. A.S. Tanenbaum, M.F. Kaashoek, and H.E. Bal, ''Parallel Programming using Shared Objects and Broadcasting,'' *IEEE Computer* (1992, to appear).
12. H.E. Bal, *Programming Distributed Systems*, Prentice Hall International, Hemel Hempstead, England (1991).
13. D.K. Gifford, ''Weighted Voting for Replicated Data,'' *Proceedings 7th Symposium*

Operating Systems Principles, Pacific Grove, CA, pp. 150-162, ACM SIGOPS (Dec. 1979).

14. P.A. Bernstein and N. Goodman, ‘‘Concurrency Control in Distributed Database systems,’’ *ACM Comping Surveys* **13**(2), pp. 185-221 (June 1981).
15. T.A. Joseph and K.P. Birman, ‘‘Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems,’’ *ACM Transactions on Computer Systems* **4**(1) (Feb. 1987).
16. R. van Renesse and A.S. Tanenbaum, ‘‘Voting with Ghosts,’’ *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, CA, pp. 456-462 (June 1988).
17. S.B. Davidson, H. Garcia-Molina, and D. Skeen, ‘‘Consistency in Partitioned Networks,’’ *ACM Comping Surveys* **17**(3), pp. 341-370 (Sept. 1985).
18. S.J. Eggers and R.H. Katz, ‘‘A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation,’’ *15th International Symposium on Computer Architecture*, Jerusalem, Israel, pp. 373-382 (May 1989).
19. K. Li and P. Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *ACM Transactions on Computer Systems* **7**(4) (Nov. 1989).
20. S.E. Lucco, ‘‘A Heuristic Linda Kernel for Hypercube Multiprocessors,’’ *Conf. on Hypercube Multiprocessors*, pp. 32-38 (1987).
21. M.F. Kaashoek and A.S. Tanenbaum, ‘‘Group Communication in the Amoeba Distributed Operating System,’’ *11th Int’l Conf. on Distributed Computing Systems*, Arlington, Texas, pp. 222-230 (20-24 May 1991).
22. A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum, ‘‘Experiences with the Amoeba Distributed Operating System,’’ *Comm. ACM* **33**(2), pp. 46-63 (Dec. 1990).
23. M.F. Kaashoek, R. van Renesse, H. van Staveren, and A.S. Tanenbaum, ‘‘FLIP: an Internet Protocol for Supporting Distributed Systems,’’ Report IR-251, Vrije Universiteit, Amsterdam, The Netherlands (June 1991).
24. B. Liskov, ‘‘Distributed Programming in Argus,’’ *Communications of the ACM* **31**(3), pp. 300-312 (March 1988).
25. L. Lamport, ‘‘Time, Clocks, and the Ordering of Events in a Distributed System,’’ *Communications of the ACM* **21**(7), pp. 558-565 (July 1978).
26. J.-F. Jenq and S. Sahni, ‘‘All Pairs Shortest Paths on a Hypercube Multiprocessor,’’ *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, Ill., pp. 713-716 (Aug. 1987).
27. S. Owicki and A. Agarwal, ‘‘Evaluating the Performance of Software Cache Coherence,’’ *Proceedings 3nd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, pp. 230-242 (April 1989).
28. A.L. Cox and R.J. Fowler, ‘‘The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experience with PLATINUM,’’ *Proceedings 12th Symposium Operating System Principles*, Litchfield Park, AZ, pp. 32-44, Rochester (Dec. 1989).
29. M.L. Scott, T.J. Leblanc, and B.D. Marsh, ‘‘Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System,’’ *International Conference on Parallel Processing*, St. Charles, Ill., pp. 255-261 (Aug. 1988).
30. J.D. Noe, A.B. Proudfoot, and C. Pu, ‘‘Replication in Distributed Systems: The Eden

Experience,'' TR-85-08-06, Dept. of Computer Science, University of Washington, Seattle (Sept. 1985).

31. J.H. Morris, M. Satyanarayan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith, ''Andrew a Distributed Personal Computing Environment,'' *Communications of the ACM* **29**(3), pp. 184-201 (March 1986).
32. J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch, ''The Sprite Network Operating System,'' *IEEE Computer* **21**(2), pp. 23-37 (Feb. 1988).
33. W.G. Levelt, M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum, ''A Comparison of Two Paradigms for Distributed Shared Memory,'' *Software—Practice and Experience* (1992, to appear).
34. S. Ahuja, N. Carriero, and D. Gelernter, ''Linda and Friends,'' *IEEE Computer* **19**(8), pp. 26-34 (Aug. 1986).
35. M.F. Kaashoek, H.E. Bal, and A.S. Tanenbaum, ''Experience with the Distributed Data Structure Paradigm in Linda,'' *First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL, pp. 175-191 (Oct. 1989).