Using Replication and Caching to Achieve Low Read and Write Latencies in Distributed Objects

Hann Wei Toh
hannwei@computer.org
+65 792 1563

Hinny Pe Hin Kong
ephkong@ntu.edu.sg
+65 790 5438

Mohammed Yakoob Siyal
eyakoob@ntu.edu.sg
+65 790 4464

School of Electrical and Electronic Engineering
Nanyang Technological University
Nanyang Avenue
Singapore 639798
(Mails should be directed A/P Hinny Pe Hin Kong)

Abstract

Read and write operations in distributed objects have traditionally been exposed to network latencies. This results in such invocations being expensive in cases where network latencies are high. Temporal locality has been exploited in several products through the use of object mobility to achieve read and write time reduction. But such an approach hinges latency reduction for both reads and writes on temporal locality.

We studied the replication and caching strategy employed by several distributed shared memories and implemented it using The Common Object Request Broker Architecture (CORBA), a contemporary distributed object system. Together with a variant of relaxed consistency property, which ensures the correctness of program execution in the presence of object replicas, and a dynamic semi-distributed manager ownership scheme, which enforces the single-writer constraint and hence write serialization, the strategy eliminates the dependency of read latency reduction on temporal locality. It also shields read times from the effect of frequent remote writes, each of which triggers a local state update.

Write latency reduction is dependent on the property of temporal locality. But results obtained show that a small number of nodes, coupled with a small (invocation interval/network latency) ratio, could produce the same effect in the absence of the property. Further, a small ratio of invocation intervals also helps reduce write latencies experienced by nodes with shorter intervals.

Keywords: latency, distributed objects, replication, caching, consistency

1 Introduction

Read and write operations in distributed objects have traditionally been exposed to network latencies, resulting in such invocations being expensive in cases where network latencies are high. In distributed shared memory systems, a predecessor of distributed objects, replication and caching have been used to take advantage of *temporal locality*[1] to bring data close to individual nodes where it is used, thereby shielding the nodes from high data access latencies. Such approaches, however, are not common[2] in contemporary distributed object systems[3]. The mobile object

[1] The property of a system in which recently accessed items tend to be accessed again in the near future.

[2] Replication alone is common in distributed systems, including distributed objects. It is used primarily to achieve fault-tolerance, and sometimes also load-balancing [21]. Examples include Isis, Horus, Ensemble, Maestro [6, 4, 3, 5, 33, 14] and Legion [13] etc. But how the presence of both replication and caching might affect read and write latencies in distributed objects, and the corresponding performance characteristics in terms of read and write times have largely not received coverage.

[3] CORBA [24, 28], DCOM [20, 19] and Java [12] etc.

approach [21] has instead been adopted by several products to enable fast repetitive invocations. Though effective in enabling low latency repetitive reads and writes, this method relies heavily on the property of temporal locality of applications. The absence of the property will result in both of them being unachievable.

We designed and implemented an architecture based on the distributed shared memory approaches that not only provides low latency reads and writes, but also eliminates the dependency of faster reads on the property of temporal locality. The results obtained show that while the time taken to perform a write operation varies depending on how frequent it is executed and on the number of concurrent writes, reads demonstrate a consistent behavior with a relatively constant low invocation time, regardless of the frequency of local state updates induced by remote writes.

An immediate application area of such an architecture is the implementation of an object-based distributed shared memory (DSM) system that utilizes contemporary distributed objects. A number of research projects have been conducted to discover ways of reducing the latency of memory operations [2, 11, 10, 27] and improving the heterogeneity support [36, 7, 35] of this group of systems. Memory latency reduction is mainly achieved through relaxed memory consistency models, which exploit the possibility of reordering of memory operations and the relaxation of the write atomicity requirement, while still maintaining the correctness of program execution. Heterogeneity support improvement, on the other hand, deals with the interoperability of different hardware, which may have distinct data formats and granularities, to ensure that they can function together as a single DSM system.

Some research has been done to integrate the two properties together, as in Mermaid [36], Agora [7] and Java/DSM [35]. Among these systems, Java/DSM appears to be the most promising due to its additional capability of shielding hardware differences from programmers. However, though it provides seamless heterogeneity support through the use of homogeneous Java objects, it is built on the foundation of a hardware dependent page-based DSM system, TreadMarks [16].

One of the main reasons that led to the use of TreadMarks, instead of RMI[4], in Java/DSM is that the former offers built-in replication and caching capabilities which reduce latencies of method invocations on remote objects. By integrating these capabilities into distributed objects, we eliminate the reliance on a less widely supported underlying DSM infrastructure. At the same time, this approach unifies the representation of memory locations, or data items, and the mechanism used to maintain replica consistency, implementing both of them in distributed objects. While it does not necessarily lead to a higher degree of simplicity, it does help reduce the maintenance effort required and enhance portability, since there is only a single distributed object infrastructure to take care of, and it generally exhibits uniformity in data representation and programming interface on different platforms.

The sections that follow provide a discussion on the related work, a more detailed description of our architecture, as well as its consistency properties and protocols. The dynamic semi-distributed manager ownership scheme adapted from the page-based DSM, Ivy, is then presented. These are followed by results showing the read and write performance of the architecture. A discussion section then explores further the possible performance enhancement techniques.

2 Related Work

Here we cover related work in the fields of distributed objects, fault-tolerant distributed systems and object-based DSMs. These include the recent distributed object product Voyager, fault-tolerant distributed system Maestro, object-based DSMs Linda, Orca and Java/DSM, and a CORBA-compliant object caching system Flex.

2.1 Voyager

Voyager [25, 26] from ObjectSpace uses the mobile object approach [21] to reduce object method invocation times, or read and write latencies we refer to in our work. A unique object may be transferred to a location close to a client upon request. As the network distance between the object and the client is reduced, subsequent method invocations consume less time. JumpingBeans [1] from Ad Astra Engineering employs the same technique to achieve the objective. But for this approach to be effective in reducing the overall invocation time, method invocations on the object need to exhibit the property of temporal locality. That is, the calls from the client on the same object

[4] The Remote Method Invocation of Java

need to be made in the near future. Otherwise, another client might request the object to be transferred elsewhere, and the next invocation by this client on the same object will have to trigger the transfer process again, and thus will not observe time reduction. This applies to both read and write invocations. The use of replication and caching in our architecture eliminates this necessity for read operations, as the replica cached locally is able to serve read requests without making a remote call.

Voyager has a similar facility based on replication and caching. In its implementation of group communications, objects in the system reside in a logical *space*. A space is composed of *subspaces*, each of which resides in a node. Objects may be replicated and cached in different subspaces. Each subspace may have a multicast proxy[5] that serves as an entry point for the dissemination of invocation requests to replicas in the space. A propagation method is used to multicast the requests, with each subspace propagating the requests that it receives to neighboring subspaces, which are connected through the explicit invocation of a Voyager-specific function[6]. This facility helps to reduce method invocation times. However, it does not preserve the order of requests. Replicas in different subspaces do not always receive multicast messages in the same order. Further, the order of messages received by replicas may not match the order in which they were sent. These result in replicas in different subspaces not necessarily stabilize at the most recent states. In our architecture, we added time-stamps and an ownership scheme to ensure the observance of the same execution order among replicas, and the preservation of program order among writes, to achieve the consistent stabilization of replica states.

2.2 Maestro

Maestro [34], though developed primarily to assist the fault-tolerance research [6, 8] in Cornell University and not intended to offer low latency method invocations, is discussed here due to the resemblance between its object replication approach and our replication and caching strategy. Object replication in Maestro is done based on the concept of *active replication* [4]. *Totally ordered multicasts* is used to ensure all replicas within the same Horus [33] or Ensemble [14] *group* receive requests in the same order, thereby maintaining consistent object states among the replicas.

As will be seen in later descriptions, our architecture depends on state update propagations instead to keep replicas consistent. This method allows older states to be discarded if they arrive at a replica later than a more recent state. Unnecessary system stalls may be avoided, as a replica does not have to wait for an old state to arrive if it has received a more recent update request through the non-order preserving state delivery mechanism. However, the system will still stabilize at the most recent states consistently.

2.3 Linda

The object-based distributed shared memory system, Linda [31, 9], is built on the foundation of the concept of *tuple space*. A tuple space appears as a global shared memory to programmers. *Tuples*, similar to data objects without user-defined behaviors, or structures in C language, are stored in a tuple space. The tuple space may be composed of several subspaces, each of which may reside on separate machines. A unique tuple may be stored in any one of the subspaces. It can be created in one subspace, and subsequently located through broadcasting and matching of templates. Replication of a tuple in multiple subspaces may help reduce the access time by spreading the load more widely and by taking advantage of locality. A two-phase commit protocol prevents races [31].

The handling of data items, or tuples, in Linda which relies on a fixed set of primitives like *in*, *out* and *read*, differs from that of contemporary distributed objects like Java and CORBA that allow the specification of user-defined methods. The former offers a simpler and clean interface. While the latter, on which our architecture is based, enables the coupling of additional user methods through the adoption of more complicated architectures. However, it should be possible to implement various memory consistency models in Linda, besides a two-phase commit protocol, in order to ensure correctness in the existence of replicated tuples. This will give rise to an architecture similar to ours in functionality, offering low access latencies while ensuring correctness.

[5] Created with a call to *getMulticastProxy* defined in the interface *ISubspace*.
[6] The method *connect* defined in the interface *ISubspace*.

2.4 Orca

Orca [31] is developed using the Amoeba system [31]. It features a compiler and a runtime system, as well as a language without pointers and which does runtime checking of array bounds. Both unique and replicated objects are supported. The latter reduce read and write times. The Orca compiler inserts code into an application to help the runtime system determine the locality of objects and the need to acquire locks etc. New values or parameters are delivered to maintain consistent states among replicas, that is, an update-based algorithm is used, rather than relying on invalidations. The position of synchronization determines the level of consistency of the system, which ranges from sequential to entry and lazy release consistencies. For all levels, either a sequencer or a primary copy is used to maintain the consistent order of invocations visible to all replicas.

An update-based Orca system that transfers states, instead of parameters, is similar to our architecture. The difference between the two systems lies in the underlying infrastructure. Orca uses its own language and runtime, while our architecture utilizes the language independent CORBA, or alternatively Java, if equivalent services of the latter are used instead. The current effort has been concentrated on the investigation of read and write performance, so the consistency level supported by our architecture is rather limited, with only a single form of relaxed consistency. Details of the consistency properties and protocols [32] are presented in Section 4.

2.5 Java/DSM

Java/DSM [35] utilizes Java as the programming language. It uses a group of modified Java virtual machines, which run on TreadMarks [16], a hardware dependent page-based DSM system. The virtual machines offer a homogeneous parallel processing environment, with identical data formats and granularities across different physical machines, though they themselves are hardware dependent implementations. Java objects are replicated in the virtual machines to provide reference locality, which helps reduce the latency of memory operations. The underlying DSM system offers memory module communications support, ensuring that the states of the replicated objects are consistent.

Both our architecture and Java/DSM use replication to achieve reference locality. This approach is also similar to the general shared memory abstraction mentioned in [10] which assumes each processor, or node in our case, maintains a complete copy of the memory. The replication of objects in each node, instead of having unique objects in the system accessible only through remote invocations, is the major difference between these two systems and conventional distributed object systems like Java and CORBA.

We chose to implement the DSM functionalities and consistency properties in distributed objects. Though both TreadMarks and distributed object systems are basically hardware and operating system dependent implementations, the latter have gained wider support, largely due to their broader area of application. This may enhance the heterogeneity support of the DSM system. The difference resulted is that while Java/DSM relies on page-based consistency maintenance, our system maintains consistency at the object level.

2.6 Flex

Flex [17] is a CORBA compliant object caching system that achieves latency reduction through the caching of replicated objects. It has two consistency policies, causal and strong. Object consistency is maintained through the mutual consistency and scalable time-stamping mechanisms. It relies on the creation and validation time-stamps, as well as information like access type, in determining if objects are consistent. The state of each object determines the view of the node that houses it. This is the node view. Node views in turn form the system view, which is the state of the entire system. Objects are updated incrementally through communications among nodes.

The system features replicated CORBA objects, which have homogeneous data type and format on heterogeneous hardware and operating systems. They shield the programmers from hardware differences, while introducing language independence into the system, enabling codes written in different languages to run in the same system. This and its consistency maintenance capability are properties similar to that of ours. However, the system does not have specified behavior for read and write operations, as well as mechanisms supporting atomic read-modify-writes. These are explicitly specified in the consistency model of our architecture to achieve a close satisfaction of the shared memory consistency properties documented in [10].

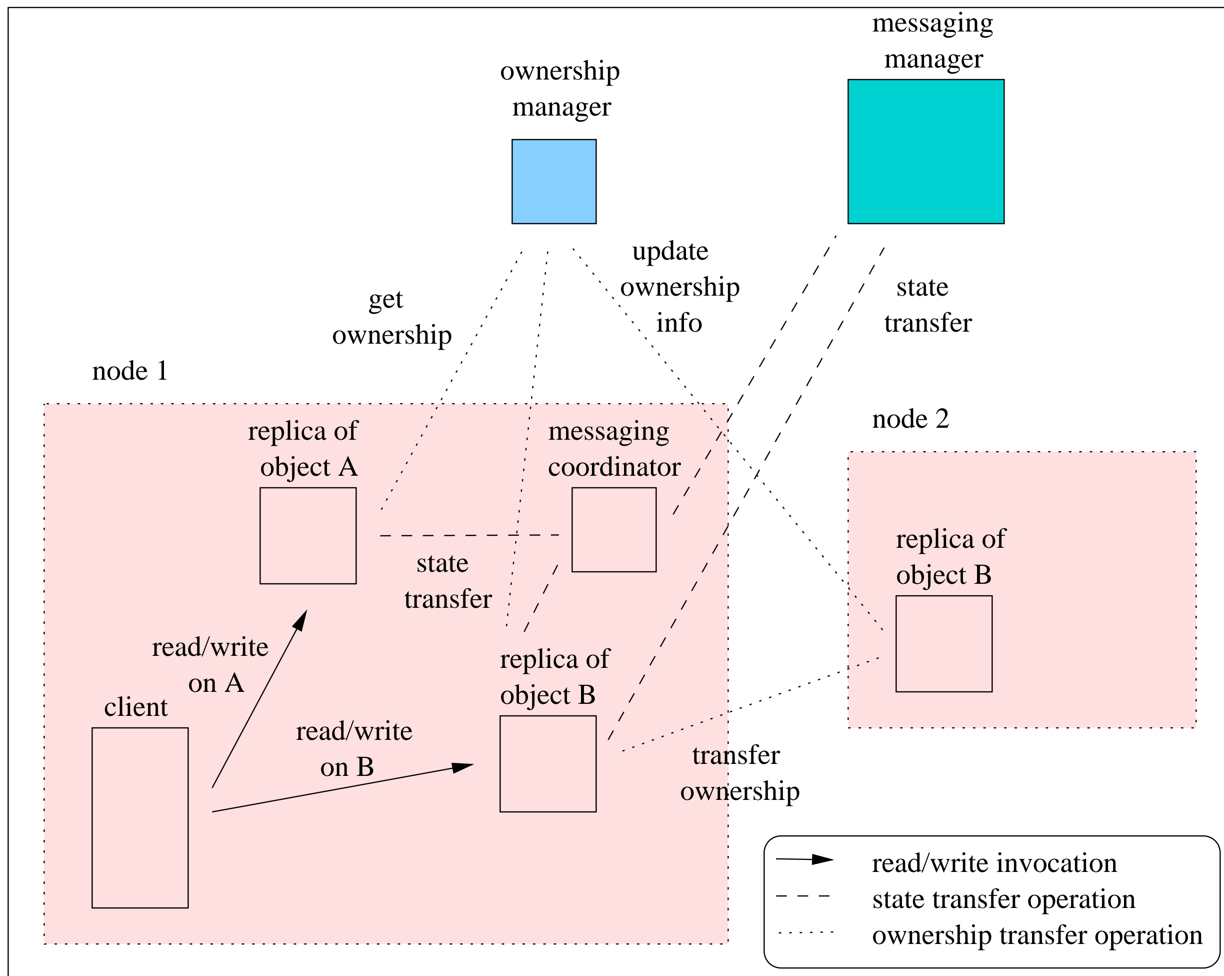

Figure 1: Architecture Overview

3 Architecture Overview

The current version of the architecture is designed with CORBA [24] and utilizes the standard naming [23] and event [22] services. It consists of four major components: an ownership management module, a module handling state transfers, server object replicas and clients. Figure 1 shows the components and their interactions with each other. In the figure, the ownership management module is represented by its implementation name of ownership manager. The module handling state transfers, on the other hand, is represented collectively by the messaging manager and a group of messaging coordinators. Server object replicas and clients are represented with replicas and clients respectively.

The ownership management module is responsible for the transfer and updating of ownership information. The module handling state transfers takes care of the delivery of new states to server object replicas. Server object replicas are the replicas of server objects in the system. While clients are programs that perform tasks with the aid of communications with other clients through accessing the server object replicas.

The concept of ownership is used in the architecture to serialize writes to replicas of the same server object. For a server object, only one owner is present in the system at any instant. Owners are nodes, which are described below, that hold the ownership of certain server objects. Only the owner of a server object may perform a write operation on the object. The transfer of ownership among nodes is handled by the ownership management module. The mechanism involved in the transfer is termed an *ownership scheme*. Sections 3.2.1 and 5 contain details of the application and design of our ownership scheme.

Server object replicas and clients are grouped into *nodes*. Only one replica of a server object may reside in a node, though each node may have replicas of different server objects. Multiple nodes may be housed in the same physical machine, or components of a node may be spread across several machines. But, typically, each machine will house one node.

The following is a description of the handling of read and write operations in the architecture. Details of the consistency properties and the respective protocols used to achieve them are described in Section 4.

3.1 Reads

When a client performs a read operation on a server object, the corresponding method on the replica in the same node is invoked. The current state of the replica is returned immediately. This results in a short invocation time for read operations, and eliminates the dependency of low read latencies on temporal locality, which is present in JumpingBeans [1] and Voyager [25] in non-replication mode.

3.2 Writes

3.2.1 Ownership Transfer

Write operations, however, rely on the ownership status of a node. If a node is the owner of an object, a write initiated by its client may update the state of the corresponding local replica directly. This enables a local write to complete immediately. If the node is not the owner, it will have to request for ownership from the owner node, which is the node currently holding the ownership.

We adapted the dynamic distributed manager ownership scheme of Ivy [18], a page-based distributed shared memory system, to our architecture. The concept of *probable owner*[7], which was first introduced in Ivy and later used in Munin [32, 31] and Midway [31], is implemented. However, instead of having a probable owner contacting the next probable owner with one-way messages, remote method invocations are used. The ownership requester performs a remote method invocation on each probable owner until the true owner is reached. When contacted by the requester, the true owner releases the ownership. Then both of them inform other nodes about the identity of the new true owner. Details of the ownership scheme, which we named dynamic semi-distributed manager ownership scheme, is presented in Section 5.

The architecture does not use hardware dependent broadcast and multicast facilities directly. The CORBA event service is utilized in ownership information dissemination. The implementation of the service may be based on either standard remote method invocations or multicasts [15][8]. This shields the architecture from the underlying information dissemination details, thus increasing its portability.

After the ownership is obtained, the requesting node proceeds to complete the associated write operation by updating the state of the local replica. Subsequently the write invocation returns. Future writes originating from the same node and to the same object will complete immediately, without the need to perform another ownership request, if the node is still the owner. This is similar to the approach taken by Voyager [25] in non-replication mode, in which write latency reduction relies on the property of temporal locality.

The adoption of the concept of ownership ensures writes to the same object are serialized. This is analogous to applying *global time ordering* [31] to writes on each individual replicated object, which avoids the arise of conflicts due to concurrent writes to different replicas.

[7] When a node is contacted for the ownership of an object, it releases the ownership if it is the owner. Otherwise, it returns the reference to another node which it thinks is the owner. In the latter case, the requester proceeds to contact the next node. The process is repeated until the true owner is reached. Nodes which refer the requester to other nodes are termed *probable owners*.

[8] The current CORBA specification (version 2.2) contains guidelines only for an IIOP-based event service. The IP multicast-based OrbixTalk [15] thus has a non-standard implementation. But it offers a standard event service interface.

3.2.2 State Transfer

Upon the completion of a write, the replica delivers its state to the module handling state transfers. This module operates in the same way as the ownership management module. The event service [22] is employed to disseminate the new state to all other nodes. Each write is assigned a *currentness* value, which is a long integer. A node that receives a new state compares the currentness of this new write and that of the previous write it received. If the new write has a higher currentness value, indicating a more recent state, the node updates its replica. Otherwise, the new state is discarded.

Though the current version of the architecture employs only a single event channel for state delivery, the possibility of using multiple concurrently running channels, each serving a different group of nodes, is not ruled out. This will result in nodes receiving out of order writes. However, while the state delivery mechanism does not preserve write orders, the selective acceptance of writes based on currentness values ensures replicas will observe ordered updates, thus enabling the system to stabilize consistently at the most recent states.

4 Consistency Model

The set of consistency properties and protocols of the architecture is presented below using the notations employed in the consistency model specifications in [10]. The notations are described below. For simplicity, we consider a model similar to that of the general shared memory abstraction in [10] which assumes that a complete copy of the memory is replicated in each node.

The definitions of the notations are:

R Read operation, an operation which returns the state of an object.

$R_{init}(n)$ The initial stage of a read operation, executed on node n.

$R(n)$ The subsequent stage of a read operation, executed on node n.

W Write operation, an operation which alters the state of an object.

$W_{init}(n)$ The initial stage of a write operation, executed on node n.

$W(n)$ The subsequent stage of a write operation, executed on node n.

$\stackrel{po}{\to}$ The order in which operations are arranged in a program.

$\stackrel{xo}{\to}$ The order in which operations are executed by the processors.

4.1 Consistency Properties

The consistency properties, or execution order ($\stackrel{xo}{\to}$) characteristics, of our system are as follows.

1. Read sub-operations

   A read operation R is composed of the sub-operations $R_{init}(i)$ and $R(i)$, where i is the node in which the read is performed.

2. Write sub-operations

   A write operation W is composed of the sub-operations $W_{init}(i)$, $W(1)$, ..., $W(n)$, where n is the number of replicas.

3. Atomic reads in individual nodes

   If a read operation R in node i comprises the sub-operations $R_{init}(i)$ and $R(i)$, then there is no other operation or sub-operation in node i that is ordered between $R_{init}(i)$ and $R(i)$ by $\stackrel{xo}{\to}$.

4. Atomic writes in individual nodes

   If a write operation W in node i comprises the sub-operations $W_{init}(i)$, $W(i)$ and $W(j)$, where j=1,...,n and $j \neq i$, then there is no other operation or sub-operation in node i that is ordered between $W_{init}(i)$ and $W(i)$ by $\stackrel{xo}{\rightarrow}$.

5. Return value for reads

   If there is a write W to the same object as a read R on node i, such that $W(i)\stackrel{xo}{\rightarrow}R(i)$, then $R(i)$ returns the value of the last such $W(i)$. Otherwise $R(i)$ returns the initial value of the object.

6. Write order preservation for conflicting writes

   For two write $W1$ and $W2$ on two identical object replicas, if $W1\stackrel{xo}{\rightarrow}W2$, then $W1(i)\stackrel{xo}{\rightarrow}W2(i)$for all i.

7. Consistent final states

   If three writes $W1$, $W2$ and $W3$ to the same object appear on node i as

   $W1(i)\stackrel{xo}{\rightarrow}W2(i)\stackrel{xo}{\rightarrow}W3(i)$

   the execution on another node j, where $j \neq i$, may appear in either one of these forms:

   $W1(j)\stackrel{xo}{\rightarrow}W2(j)\stackrel{xo}{\rightarrow}W3(j)$
   $W1(j)\stackrel{xo}{\rightarrow}W3(j)$
   $W2(j)\stackrel{xo}{\rightarrow}W3(j)$
   $W3(j)$

   This implies that all object replicas will have the same state when the propagation of all writes have completed, but they may not have acquired all the intermediate states when that happens.

8. Serialized conflicting writes

   If the writes $W1$ and $W2$ are directed to two identical object replicas, and that

   $W1$ is composed of $W1_{init}(i)$ and $W1(i)$ on node i

   and

   $W2$ is composed of $W2_{init}(j)$ and $W2(j)$ on node j

   then the execution orders will be either

   {[$W1_{init}(i)\stackrel{xo}{\rightarrow}W2_{init}(j)$] and
   [$W1(i)\stackrel{xo}{\rightarrow}W2(j)$]}

   or

   {[$W2_{init}(j)\stackrel{xo}{\rightarrow}W1_{init}(i)$] and
   [$W2(j)\stackrel{xo}{\rightarrow}W1(i)$]}

9. Program order preservation for non-conflicting writes

   If the writes W and W' are directed to different objects on node i, and they are in program order, such that $W(i)\stackrel{po}{\rightarrow}W'(i)$, then $W(j)\stackrel{xo}{\rightarrow}W'(j)$ for all j.

10. Atomic read-modify-writes

    By definition, an atomic read-modify-write is represented by $R\stackrel{po}{\rightarrow}W$. If there are two such operations, $R1\stackrel{po}{\rightarrow}W1$ and $R2\stackrel{po}{\rightarrow}W2$, on identical object replicas on nodes i and j respectively, then the execution is either

    {[$R1\stackrel{xo}{\rightarrow}W1$] and
    [$R2\stackrel{xo}{\rightarrow}W2$] and
    [$W1\stackrel{xo}{\rightarrow}R2$]}

or

{[$R1 \stackrel{xo}{\rightarrow} W1$] and
[$R2 \stackrel{xo}{\rightarrow} W2$] and
[$W2 \stackrel{xo}{\rightarrow} R1$]}

on all nodes.

4.2 Consistency Protocols

The protocols used to achieve the consistency properties are as follows. Each item is numbered according to the corresponding property in the previous section.

1. The differentiation of $R_{init}(i)$ and $R(i)$ is done solely for demonstrating the atomic nature of reads. They may represent the beginning of a read invocation request handling and the return of the state respectively. The notations also imply that a read is visible only in the node where it is performed. Other nodes are not affected by the read.

2. After a write invocation has been made on a replica, the replica delivers its encoded new state to the coordinator in the local node, which subsequently sends the state to an event channel [22]. The event channel disseminates the state to the coordinators of all other nodes. Upon receiving the state, a coordinator delivers it to the target replica residing in the same node as the coordinator. The replica then decodes the state and uses it to update itself. So, besides $W_{init}(i)$ and $W(i)$ on the initiating replica, all other replicas also execute the same $W(j)$ where j=1 to n and $j \neq i$. n is the number of nodes with such replicas.

3. A read operation corresponds to an invocation on a CORBA or similar distributed object replica which returns the state of the replica. There are no distinct components within the invocation that can be labeled as $R_{init}(i)$ and $R(i)$. But we may view the beginning of invocation request handling as $R_{init}(i)$, and the return of the state as $R(i)$, if precise definitions of the two sub-operations are required. For a properly synchronized or thread-safe object implementation, we expect the read operation to be atomic. Or it should at least give the illusion of being atomic (Applicable to platforms that support different read and write locks. On such platforms, multiple reads may be performed concurrently after read locks are acquired. This makes the reads non-atomic.) by returning a valid state, which is the initial state, the state between two consecutive writes, or the state after the final write.

4. A write operation corresponds to an invocation on a CORBA or similar distributed object replica which changes the state of the replica. Similar to reads, the beginning and end of the write invocation may be viewed as $W_{init}(i)$ and $W(i)$ respectively. A write is not performed concurrently with any other read or write to the same object replica in the same node. This is achieved through proper synchronization or thread safety of the object implementation. Such a non-concurrent property implies that, in a single node, writes to the same object replica are atomic.

5. Since read and write invocations on the same object replica within a node are performed sequentially, a read always returns the value written by the last write that has completed in the node. The write invocations referred to here include both writes initiated by this node, and update requests disseminated by the event channel due to writes started in other nodes.

6. All writes are given a *currentness* value, which is a long integer showing the order of each write with respect to the others. Writes to identical object replicas are assigned a sequence of continuous currentness values. A new write is given a value equal to the value for the previous write plus one. For example, if $W11$ and $W12$ are two consecutive writes to object 1, and $W21$ and $W22$ are consecutive writes to object 2, with the second write occurring after the first, then the currentness values are as follows.

   $currentness_{W12} = currentness_{W11} + 1$
   $currentness_{W22} = currentness_{W21} + 1$

When a replica receives a request from the local coordinator to update its state, it checks the currentness value embedded in the request. If the value is larger than that of the previous update, indicating the arrival of a more recent write, it accepts the request to update its state, otherwise it ignores the request. In this way, writes to the same object are carried out in the same order in all nodes.

7. The transfer of ownership, which allows a write operation to be performed on an object replica in a node, and the transfer or dissemination of new states, or update requests, are carried out independently. So, a node might have obtained the ownership and completed a write operation locally, before the state of the previous write arrives. When the state arrives, it is ignored due to its smaller currentness value. This leads to a situation in which a write in the execution sequence is absent from this node, but present in another node. However, if there is a period during which no writes occur to any replica of the object, all replicas will receive the latest state, indicating that the effect of the latest write will be visible to all replicas eventually. This ensures replicas will have a consistent final state. Such a characteristic is adopted to enhance the system speed, so that the execution involving a node will not be stalled for a previous write to arrive, while it is ready to perform a new write.

8. We adopted the single-writer approach in managing concurrent writes. A replica can proceed with a write invocation only after it has obtained the ownership of the object. And within the system, there is only one owner for identical replicas. These ensure that writes to identical replicas are serialized. We chose to use the dynamic distributed manager ownership scheme introduced in [18] which offers better performance and higher scalability compared to centralized schemes. However, we do not use a multicast service directly to disseminate new ownership information in our system. Instead, the event service [22] in CORBA is utilized[9]. The event service may use hardware dependent multicasts, as in OrbixTalk [15], or ordinary remote method invocations, as the message dissemination mechanism. This gives the system a higher degree of portability, as multicast is still not supported by some networks and computing devices [30].

9. Each write is also assigned a time-stamp, besides a currentness value. The time-stamp records the time of the clock of the machine where the node resides when the write is performed. When the coordinator of a node receives a list of writes, or state update requests, it reorders those from the same remote node according to their time-stamp values. The writes are then dispatched to the corresponding replicas in the same order as that during execution in the originating node. This mechanism is needed as writes may arrive out of sequence due to the non-order preserving state transfer process. Currently a simple reordering algorithm is used. The coordinator stores the incoming writes in a buffer, and reorders and dispatches them based on their ages and according to their originating nodes. This method ensures that all writes from the same node are made visible to other nodes in the same order.

10. In the system, when an object replica has successfully acquired the ownership, it increments the currentness value associated with the ownership by one. After the write is performed, the currentness of the replica is updated to this new ownership currentness value. The same value is then associated with the writes, or state update requests, disseminated to other nodes through the event channel. Synchronization objects implement such a property in a unique manner with these two methods:

    wait() {
    repeat {
    while object state == 0, do nothing
    get ownership
    if currentness of ownership == currentness of replica, then set object state to 0, increment currentness of ownership, and update replica currentness
    else pause for a while}
    until the (currentness of ownership == currentness of replica) condition above is satisfied }

    post() {
    set object state to 1 }

    wait() offers an atomic read-modify-write capability to the object, allowing only one replica to change the value of the object at any one time.The synchronization objects can be used to enclose a critical section. The

[9]We added a manager to handle the event service. Because of this, we call our scheme a dynamic semi-distributed manager ownership scheme.

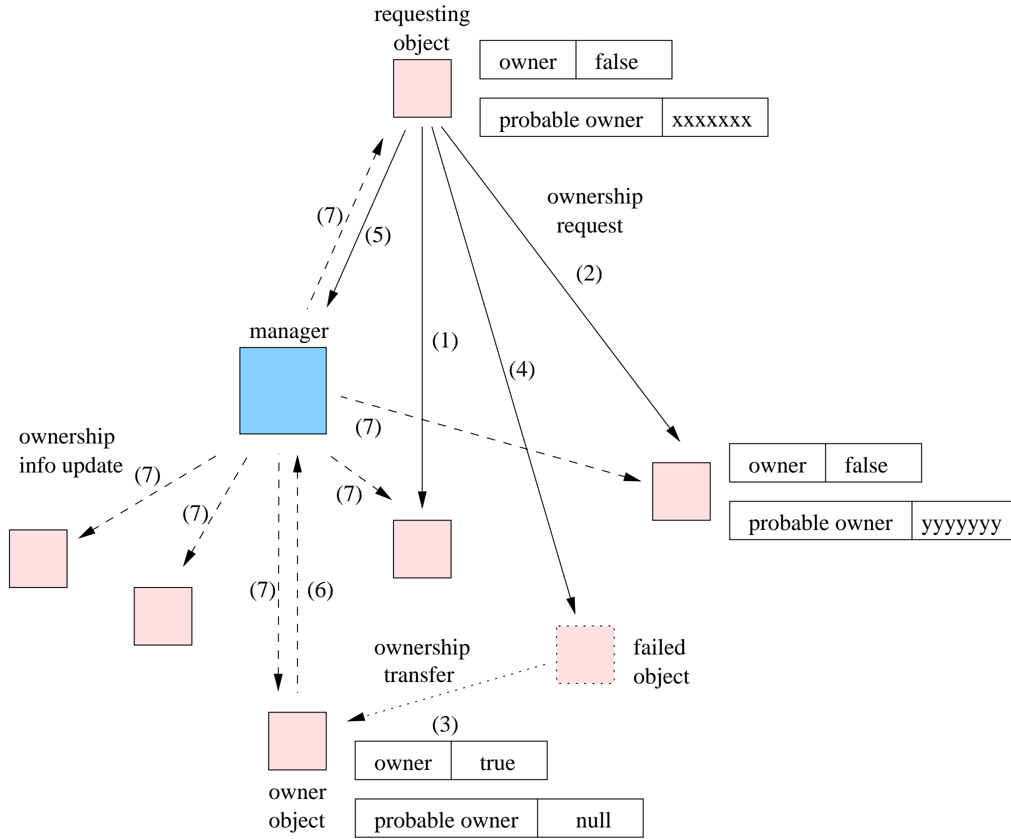

Figure 2: Dynamic semi-distributed manager scheme

object state of 0 indicates that a node somewhere in the system has entered the critical section. 1 indicates that the node has left the critical section, and now other nodes are free to compete to enter the section.

5 Dynamic Semi-Distributed Manager Ownership Scheme

5.1 Architecture

As in the case of the dynamic distributed manager ownership scheme of Ivy [18] where each individual processor maintains its own page table that contains information about the probable owner of a page, this implementation in CORBA, as shown in Figure 2, also features nodes with server object replicas that keep track of the probable owners. The *probable owner* in this implementation refers to the replica that is likely to be having write access to its own state. Similar to the DSMs case, the probable owner in this scheme may contain information about another probable owner. Eventually the sequence will terminate at the true owner. Each replica keeps information about its ownership status in its *owner* attribute. This is similar to that in the centralized manager schemes. The attribute may have a value of either *true* or *false*, indicating whether or not the replica has write access to its state. Whenever a replica obtains the ownership, it distributes messages to all other replicas cached in other nodes, informing them to update their probable owner attribute to point to itself. Besides, when an ownership transfer occurs, the replica that releases the ownership updates its probable owner attribute to point to the new owner.

We term this implementation a dynamic *semi-distributed* manager ownership scheme, since a manager is used to distribute probable owner update messages. We do not tread the path taken by DSMs, in which a processor sends update messages to individual processors or disseminates the messages with a broadcast mechanism. The former approach is slow and takes up much resource at the server, thus is not suitable for a large number of nodes. The

latter requires the existence of a broadcast facility, which though may be common in hardware within a restricted area, is not guaranteed to be accessible in a CORBA environment that is usually expected to span a broad area which may consist of individual networks with no broadcast support. Instead, this scheme uses a manager to do the task. The manager uses the CORBA event service. Though we do not use multicasts explicitly, the availability of both IIOP[10] and IP[11] multicast-based implementations of the event service gives the scheme sufficient flexibility on the choice of the underlying messaging protocol.

The manager receives a probable owner update message from a replica, and distributes it through the event service to the other replicas, which use it to update their own probable owner attributes. This manager, though is unique within the system, is not truly a potential bottleneck. In cases where the manager in unable to distribute the update messages fast enough, the ownership requesting node simply goes through a few more probable owners before reaching the true owner. This results in a longer processing time. But the scope of the effect is limited to the replicas that are involved in frequent ownership transfers. Replicas of other objects will still observe short processing times, as their update messages, though also take a longer time to arrive, might still reach them well before the embedded new ownership information is used.

5.2 Operation

Parentheses identify steps in Figure 2

When a client invokes a write-related method on a server object, the replica in the same node checks its *owner* attribute to see if it has the ownership. If it has, then the associated write operation is performed. Otherwise, it reads the IOR[12] stored in its *probable owner* attribute. The IOR points to a replica that is likely to be the owner. The method *RequestForOwnership* on the probable owner is subsequently invoked (1). If the probable owner is holding the ownership, the method returns the boolean value *true* when it is ready to release the ownership. If it is not the true owner, the method returns *false*, together with the IOR of another probable owner and an integer that this replica keeps. The integer indicates the *currentness* of the probable owner IOR it is associated with. A large value implies a more recent probable owner IOR. The currentness is used by individual replicas to determine if they should update their own *probable owner* attributes. If the probable owner update message a replica receives contains a larger currentness value compared to that it keeps, indicating a more recent IOR, it updates its probable owner IOR and currentness value, otherwise it updates the probable owner attribute. The requesting node, upon receiving a *false* return value and new probable owner information, proceeds to invoke the *RequestForOwnership* method on the next probable owner indicated by the information (2). This repeats until the true owner is reached.

When an owner releases the ownership, it sets the value of its *owner* attribute to *false* and returns *true* for *RequestForOwnership*. The requesting replica, on the other hand, sets its *owner* attribute to *true* upon getting a *true* response from the *RequestForOwnership* call. During an ownership transfer, besides changing the *owner* attribute, the owner also updates its *probable owner* attribute to point to the requesting replica, and increments its currentness value by one. The currentness value is subsequently returned. The requesting replica, upon obtaining the ownership, updates its currentness value using that returned by the owner. Both the owner and the requesting replica then immediately invoke the method *UpdateOwnerIdentity* on the manager, passing it the new owner IOR and currentness value (6)[13]. The manager uses the same strategy as that of the replicas, that is, update its owner information only if the incoming information is more recent than that it has. The manager then distributes a probable owner update message to the replicas of the server object concerned cached in all nodes (7).

5.3 Fault Tolerance Capability

Parentheses identify steps in Figure 2

The implementation has limited fault tolerance capability, as its primary purpose is to support a latency reduction architecture. The description below explains how the fault tolerance feature operates.

[10] The Internet Inter-ORB Protocol

[11] The Internet Protocol

[12] Interoperable Object Reference

[13] In the case of no failure, both the original and new owners should have made the call. Figure 2 shows the case where failure exists.

During an ownership transfer (3), both the owner and the requesting replica notify the manager of the change in ownership. So, if either one of them were to fail during the transfer process, the manager, which is responsible for distributing the corresponding probable owner update message, will still be notified of the ownership change by the remaining party (6). In this way, the manager is also able to track the ownership, and serves to provide ownership information to newly started nodes or replicas.

If in some cases, both the owner and the requesting replica or their nodes fail before the manager can be notified of the ownership transfer, the owner searching sequence will terminate at a non-existent object (4)[14]. If this happens, the requesting replica invokes the *RequestForOwnership* method on the manager (5). The manager then waits for a short period of time. If no replica claims to have obtained the ownership, the manager grants the ownership to the requesting replica. Otherwise, if the ownership were to have been successfully transferred to a third replica when the two replicas fail (3), the third replica concerned will be able to notify the manager of the new ownership status within that period of time (6). Thus a sequence of failures does not result in the manager losing track of the ownership. When the manager is notified by the third replica concerned, which is now the new owner, it returns the ownership information to the requesting node. The requesting node then begins another sequence of owner searching from this probable owner.

We note that for this strategy, that originally aims to cope with the simultaneous failure of both parties involved in an ownership transfer, to function in cases with a sequence of failures involving the transfer of the same ownership, the period of time that the manager waits has to be long enough to cover (N - 2) ownership transfers, where N is the total number of replicas of the server object of interest in the system. The proof is shown in Appendix A.

6 Results

Simulations were performed with the C++ CORBA ORB MICO 2.2.5 [29] on a Sun Ultra 1 machine running Solaris 2.5. *nanosleep()* statements were inserted prior to making synchronous invocations on remote objects to simulate network latencies. Figures 3, 4, 5, 6, 7, 8, 9 and 10 show the average read and write times experienced by clients under various load conditions. In the three-dimensional graphs, the (read/write time = network latency) plane indicates the best or smallest read/write latency that is obtainable in the absence of replication and caching.

6.1 Best Case

The best case read and write times shown in Figures 3 and 4 were obtained under the condition where no overlapped client invocations exist. Ownership transfer occurs only when a client performs its first write operation. The client in the owner node repeats the invocations to accumulate a total of 100 writes and 100 reads, with every write followed by a read. The interval between two invocations is recorded in the figures.

The best case results indicate the performance of the architecture in situations with a very high degree of temporal locality. The almost constant read and write times of less than 10 ms show that network latency is shielded from clients in such situations.

6.2 Worst Case

If the application does not exhibit the property of temporal locality, which results in an ownership transfer for almost every write, then the average write times are longer than delays caused by network latencies in most cases, as shown in Figure 6.

Depending on the number of nodes and the interval between two consecutive invocations from the same client, the write times vary. With a small number of nodes and a short interval, write times shorter than the network latency are achievable. However, as indicated in Figure 6, if the interval approaches the network latency or longer than the latter, the performance of our architecture suffers, giving write times of two to ten times the value of network latency.

[14] Imagine there is another replica that has transferred the ownership to the replica marked *failed* in Figure 2. Both of them fail before they are able to notify the manager. But the latter is able to transfer the ownership to another replica (3) before it fails.

Network latency (ms)	Invocation interval (ms)	No of nodes		
		2	5	10
10	10	3	3	3
	100	1	1	3
	500	1	1	1
100	10	2	2	3
	100	3	4	4
	500	1	1	1
200	10	1	3	4
	100	1	1	2
	500	1	1	1

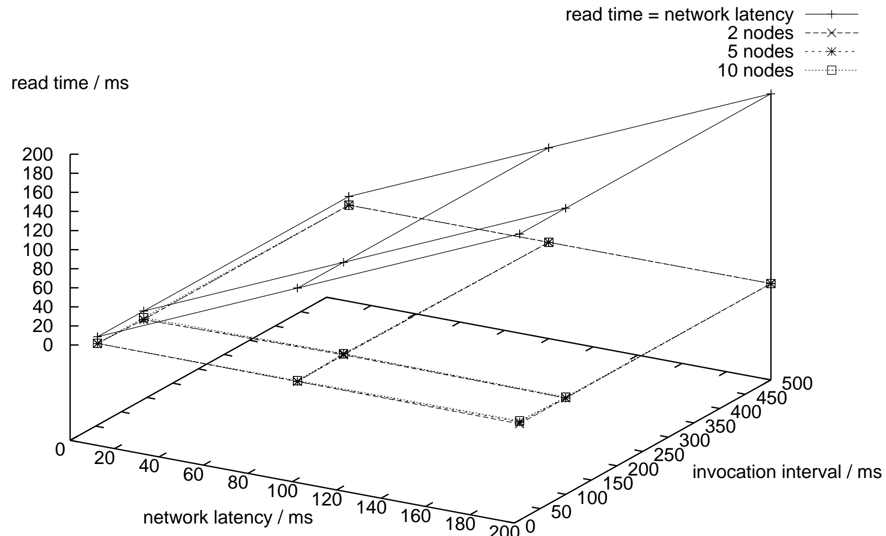

Figure 3: Best case read times (ms)

Network latency (ms)	Invocation interval (ms)	No of nodes 2	5	10
10	10	6	5	6
	100	5	5	4
	500	4	4	4
100	10	5	5	7
	100	6	7	6
	500	6	5	5
200	10	5	7	8
	100	7	6	7
	500	7	6	6

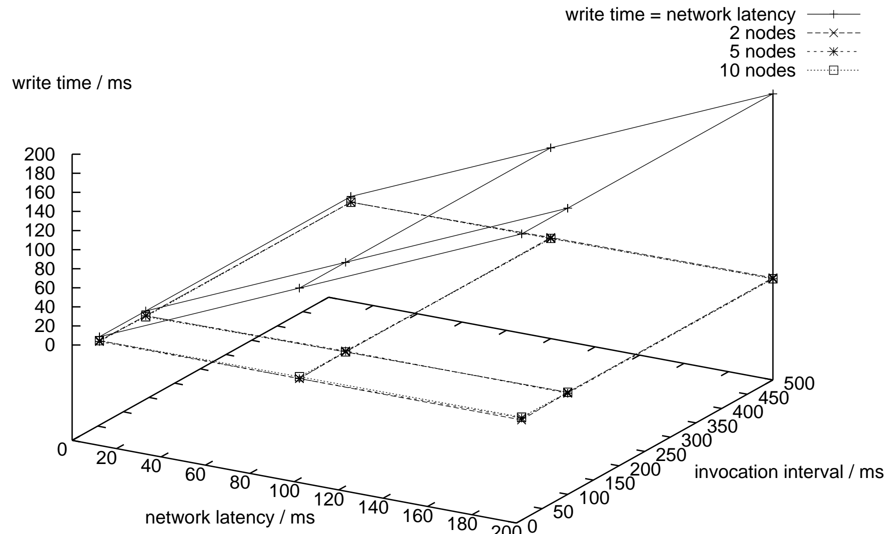

Figure 4: Best case write times (ms)

Network latency (ms)	Invocation interval (ms)	No of nodes		
		2	5	10
10	10	14	10	14
	100	2	5	17
	500	1	2	4
100	10	6	4	5
	100	6	12	11
	500	1	2	5
200	10	2	2	5
	100	1	2	4
	500	2	2	6

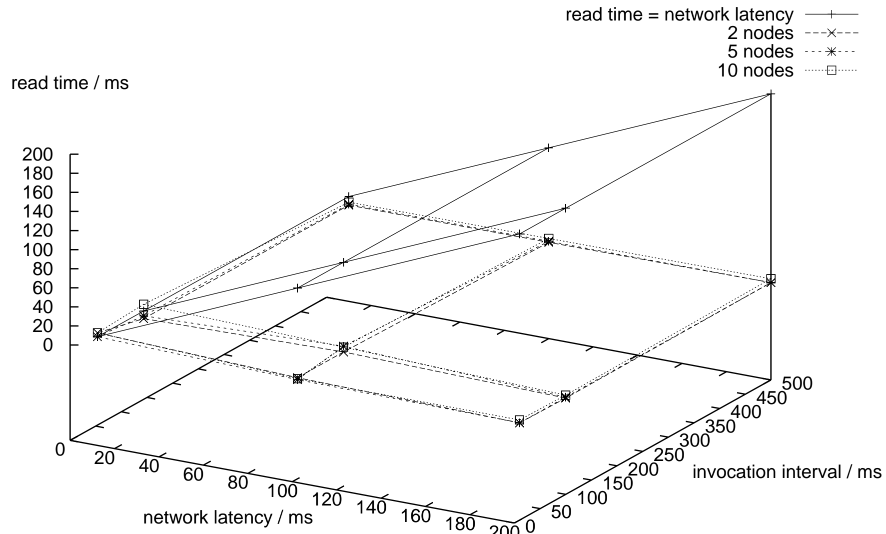

Figure 5: Worst case read times (ms)

Network latency (ms)	Invocation interval (ms)	No of nodes		
		2	5	10
10	10	88	166	377
	100	32	109	379
	500	29	32	218
100	10	50	149	328
	100	92	572	1112
	500	119	254	1205
200	10	49	133	382
	100	118	547	1446
	500	223	817	2046

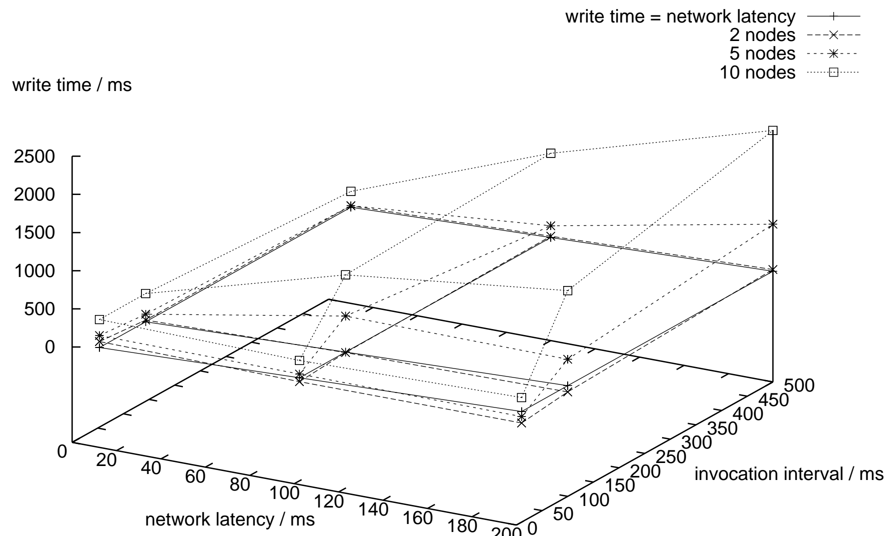

Figure 6: Worst case write times (ms)

Network latency (ms)	Invocation interval (ms)	No of nodes		
		2	5	10
10	10	7	3	4
	100	1	4	6
	500	2	3	4
100	10	3	5	4
	100	5	10	14
	500	2	3	7
200	10	2	4	11
	100	2	3	7
	500	1	2	7

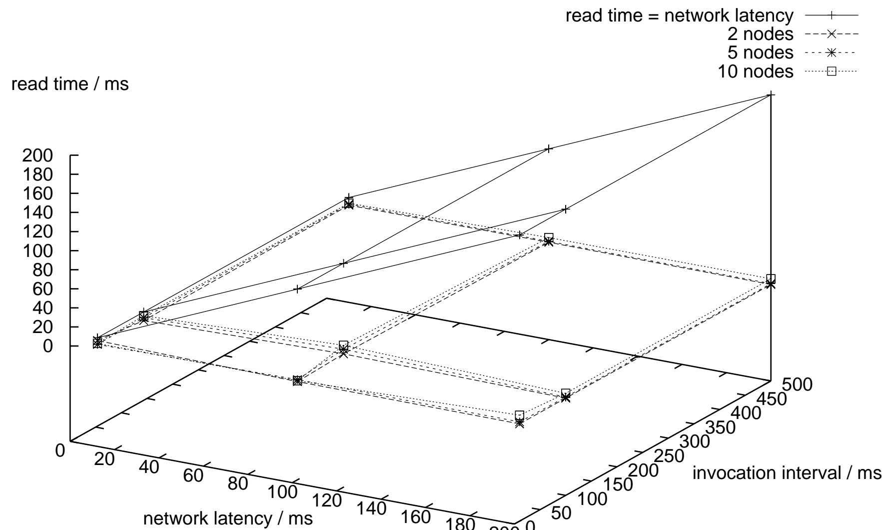

Figure 7: Moderate case read times (ms) for fixed interval node

The rare cases, where short write times can be achieved, exist mainly because short intervals between invocations help give rise to a certain degree of temporal locality.

Read times, however, remain constant in all cases, indicating the independence of read performance from temporal locality. The measurements are shown in Figure 5. Since read times do not vary much despite significant changes in the frequency of local updates induced by remote writes through state transfers, we conclude that the major factor that leads to the differences in write times is the ownership transfer overhead.

6.3 Moderate Case

Figures 7, 8, 9 and 10 show a moderate situation where the client in a node reads and writes at the intervals stated, while those in other nodes do so at random intervals of between 200 ms and 1000 ms. The results reveal that for the former to achieve write times shorter than the network latencies, the intervals between its invocations have to be around or smaller than 1/20 of the intervals of the remaining nodes. This is evident from the fact that shorter write times were achieved only if the former had an interval of 10 ms, when the remaining nodes were operating at intervals of 200 ms or longer.

Further, for the shorter write times to be achievable, it appears that the former also has to exhibit an interval of

Network latency (ms)	Invocation interval (ms)	No of nodes 2	5	10
10	10	1	3	4
	100	1	3	4
	500	2	2	4
100	10	1	3	4
	100	1	2	5
	500	2	2	4
200	10	2	2	6
	100	2	2	6
	500	3	2	7

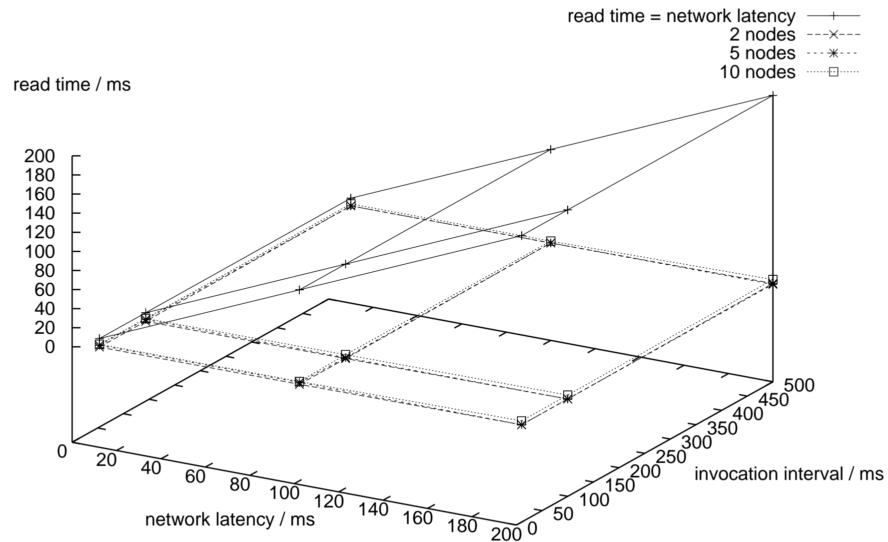

Figure 8: Moderate case read times (ms) for random interval nodes (invocation intervals belong to corresponding fixed interval nodes)

Network latency (ms)	Invocation interval (ms)	No of nodes 2	5	10
10	10	8	13	13
	100	10	33	74
	500	25	44	139
100	10	13	39	47
	100	31	151	364
	500	103	266	1092
200	10	14	39	133
	100	60	222	414
	500	175	826	2175

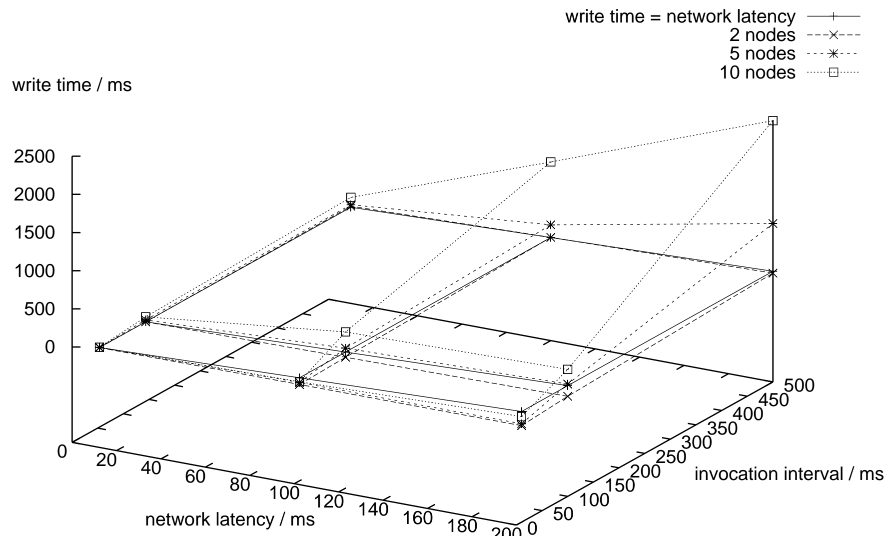

Figure 9: Moderate case write times (ms) for fixed interval node

Network latency (ms)	Invocation interval (ms)	No of nodes 2	5	10
10	10	6	43	119
	100	10	44	107
	500	26	47	130
100	10	13	234	879
	100	29	266	901
	500	103	277	1019
200	10	14	561	1804
	100	56	655	1953
	500	176	730	2055

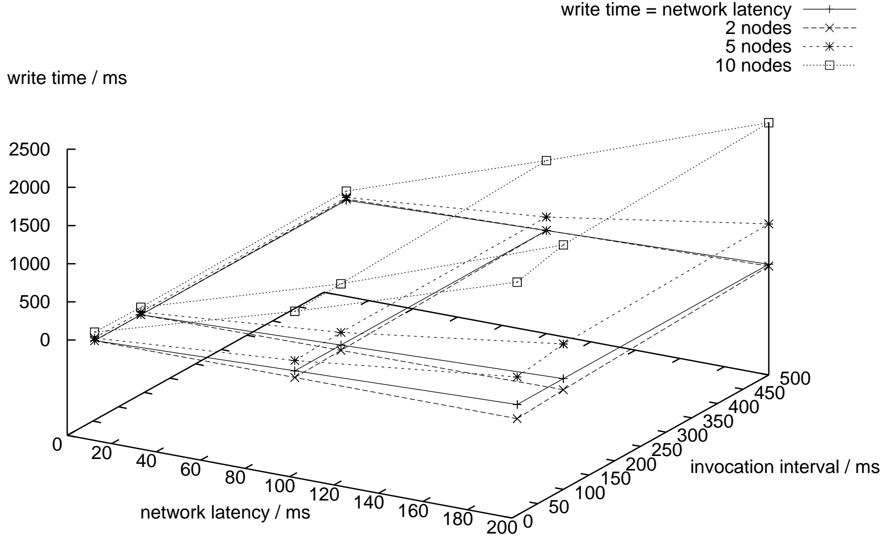

Figure 10: Moderate case write times (ms) for random interval nodes (invocation intervals belong to corresponding fixed interval nodes)

about 1/10 of the network latency. In general, this observation is in agreement with that for the worst case results, in which short write times were also achieved at small interval/(network latency) ratios.

For all load conditions, the read times remain almost constant, and stay below the network latency, as shown in Figures 7 and 8.

7 Discussions

In our work so far, we use replication and caching, together with a dynamic semi-distributed manager ownership scheme, and a relaxed consistency implementation, to arrive at an architecture with a certain set of performance characteristics for read and write times. A reduction in read time is achieved. Write time is reduced in certain cases. From the results obtained, the dependency on temporal locality for read time reduction, and in some cases of write time reduction, is eliminated. However, we acknowledge the close relationship between the extent of write time reduction and the performance of the ownership scheme, as well as write time reduction and the consistency model. Changes in the ownership scheme and the consistency model will yield different sets of performance characteristics. Here we provide an elaboration of two examples which demonstrate some possible changes.

7.1 LOCATION_FORWARD

Since our implementation of the architecture uses CORBA and IIOP, the GIOP[15] communication protocol implementation of CORBA over TCP/IP, we may utilize the GIOP LOCATION_FORWARD reply message to reduce some unnecessary marshaling and demarshaling overhead.

In the current implementation, if a *RequestForOwnership* invocation is made on a probable owner that is not the true owner, the probable owner returns the IOR of the next probable owner to the requesting replica. The ORB runtime of the requesting replica than demarshals the IOR, passes it to the replica user code, receives another *RequestForOwnership* invocation from the replica, performs the marshaling of parameters which are similar to that of the previous invocation, and delivers a new invocation request to the probable owner designated by the IOR concerned. The process repeats until the true owner is reached.

The use of the LOCATION_FORWARD reply in GIOP will eliminate the need to demarshal the IOR, pass it to the user code of the requesting replica, and receive another *RequestForOwnership* invocation from the same replica. Depending on the efficiency of the ORB runtime implementation, the processor speed, and the length of the probable owner chain, this may help reduce the time spent to obtain the ownership during a write, and hence further shortening the average write time.

This is accomplished through the use of the POA[16] and the POA policies USE_SERVANT_MANAGER and NON_RETAIN[17]. During the processing of a *preinvoke* invocation from the POA by the servant locator[18], the servant locator examines the ownership status of the target servant, which is the probable owner to which the ownership request is directed. If the target servant is the true owner, *preinvoke* returns its object reference. Otherwise, the servant locator gets the IOR of the next probable owner from the servant, and raises a *ForwardRequest* exception with the IOR as *forward_reference*. The exception is translated into a LOCATION_FORWARD reply, which the ORB runtime of the requesting replica uses to make another identical request on the next probable owner, without returning control to the replica user code.

The drawback of using this method is that we are restricted to use an ORB runtime that does not reuse location forwarding information, as the reuse of such information may result in the inability to contact a true owner subsequently. The CORBA specification[19] leaves the decision of reuse to ORB vendors. Thus, this method, though improves efficiency of writes, might not be portable across different standards-compliant ORBs.

[15] General Inter-ORB Protocol
[16] Portable Object Adaptor
[17] CORBA specification version 2.2, February 1998, in chapter 9 on The Portable Object Adaptor, pages 9-24 and 9-25.
[18] A type of servant manager
[19] CORBA specification version 2.1, August 1997, in chapter 12 on General Inter-ORB Protocol, pages 12-32 and 12-33.

7.2 Virtual Synchrony

The adoption of a different set of consistency properties will also give rise to different performance characteristics. By loosening the degree of synchrony, write times may be further reduced.

Virtual synchrony, achieved with the CBCAST protocol of Isis [5, 31], maintains only the order of causally related messages. By associating each message with a sequence number vector, and shifting the task of ordering of messages completely to receivers, writes in a virtually synchronous system suffer virtually no delay, and hence have minimal latency.

However, such performance is achievable only in a system that requires the maintenance of solely the order of causally related events. In a system similar to ours, where messages are delivered to all processes in the same order, which is achievable by another protocol in Isis, the loosely synchronous ABCAST, the write times are expected to increase substantially.

In fact, there is a technical report [8] documenting the use of virtual synchrony in designing a DSM system with three different degrees of synchrony: linearizability, sequential consistency and causal memory. However, the performance data in terms of read and write times is not available for comparison.

Another system worth noting is the pure-Java distributed computing infrastructure Novell demonstrated at the JavaOne 1999 conference. It is equipped with a client-server-based distributed consensus architecture that implements these agreement protocols: atomic broadcast, non-blocking atomic commitment and view synchrony. But details of the infrastructure, its consistency properties and performance data are not available publicly.

8 Conclusion

We reviewed several systems in the categories of distributed objects, fault-tolerant distributed systems and object-based distributed shared memories, describing the ways replication and caching have been or can be used to achieve low latency communications, or more precisely, method invocations. By producing a distributed object infrastructure with low read and write latencies, and a set of clearly specified consistency properties and protocols, we arrived at an architecture that can be a potential candidate for a portable and yet efficient distributed-object-based distributed shared memory implementation.

The protocols were proved to be effective in eliminating the dependency of low read latencies on temporal locality, and in sustaining short read times regardless of the frequency of local updates induced by remote writes. Write time reduction is apparent is the presence of temporal locality. Results show that it is also achievable if there are few nodes, which translate into lower degrees of contention for ownership, and small (invocation interval / network latency) ratios, about 1/10 in general. A node may also observe shorter write times if it has an invocation interval that is about 1/20 of that of others.

This set of performance characteristics is achieved through the use of a dynamic semi-distributed manager ownership scheme adapted from Ivy, and with the system exhibiting the specified consistency properties. Due to the dependency of performance on these attributes, changes made to the latter will result in different, possibly better, sets of performance characteristics. This was discussed with the aid of examples examining the possibility of incorporating the GIOP LOCATION_FORWARD reply and virtual synchrony.

Acknowledgment

This project is supported by grants from Nanyang Technological University, and the National Science and Technology Board of Singapore.

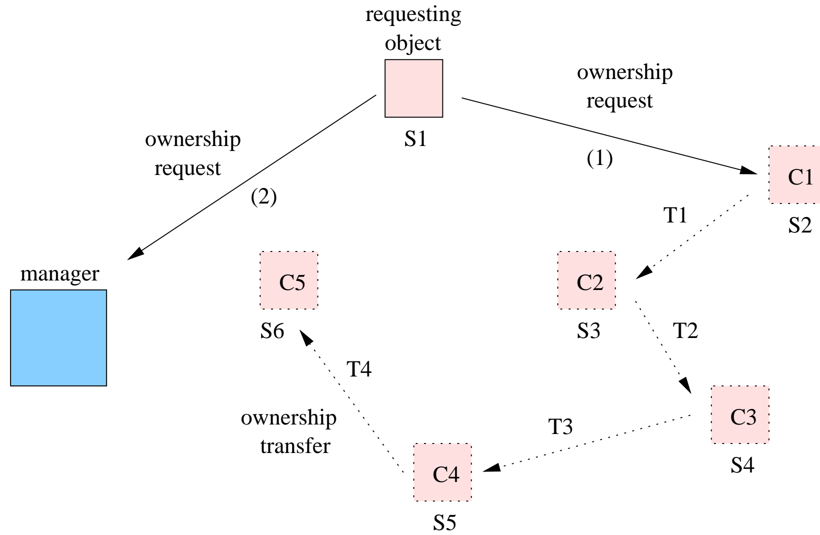

Figure 11: An example of a sequence of failures

Appendix

A Proof of manager waiting time for dynamic semi-distributed manager scheme

Figure 11 illustrates an example where a sequence of failures involving the owner occurs. There are six replicated server objects in the figure, labeled S1, S2, ..., S6. {C1, C2, C3, C4, C5} is the failure sequence. The objects marked C1, C2, ..., C5 fail in a sequence. A possible cause is that the processes housing them terminate abnormally. T1, T2, T3 and T4 are a series of ownership transfers. T1 is done successfully before C1 and C2 fail. And T2 finishes before C2 and C3 fail. The remaining transfers follow the same logic.

In this paper, we define these terms:

failure sequence A list of objects involved in a sequence of failures.

transfer sequence A series of ownership transfers that involve a failure sequence at the beginning, and end at either the same failure sequence or a surviving object.

We use

T to denote the number of transfers in the transfer sequence.

R to denote the number of objects possibly involved in the transfer sequence.

C to denote the number of objects in the failure sequence.

N to denote the total number of replicated objects.

From the figure, it is intuitive that $T = R - 1$ (note T4 and C5).

To ensure that the manager grants the ownership request from the requesting object only if no owner exists in the system, we need to determine the maximum number of ownership transfers that can happen. This maximum value is $max\ T$. Before obtaining $max\ T$, we shall compute T first. To do this, we consider two cases derived based on the number of failed objects. In both cases, we assume that all objects that fail are involved in a series of ownership transfers.

In the first case, all objects, except the one that requesting for ownership, fail. This gives $C = N - 1$. Using our assumption, we set $R = C$. So, we have,

$$
\begin{aligned}
T &= R - 1 \\
&= C - 1 \\
&= (N - 1) - 1 \\
&= N - 2
\end{aligned}
$$

In the second case, one or more than one object survives. That is, besides the requesting object, there is at least one other object that has not failed. This gives $C < N - 1$. Using our assumption, we decide that $R = C + 1$. This is so because we consider the smallest R for various Cs that is just sufficient to contain a surviving object, if there is one, that can notify the manager which is waiting for new ownership information. Thus, we have

$$
\begin{aligned}
T &= R - 1 \\
&= (C + 1) - 1 \\
&= C
\end{aligned}
$$

$$
\Rightarrow T < N - 1
$$

$$
\begin{aligned}
\Rightarrow max\ T &= (N - 1) - 1 \\
&= N - 2
\end{aligned}
$$

Combining the results obtained from the two cases, we know that

$$
max\ T = N - 2
$$

That is, the manager should wait an amount of time, which is equivalent to the time taken for $N - 2$ ownership transfers to complete, before it decides to grant the ownership request from the requesting object. This gives the system some form of fault tolerance capability by eliminating the possibility of the simultaneous existence of more than one owner for a group of replicated objects.

References

[1] Ad Astra Engineering Inc. *Jumping Beans White Paper*, August 1998.

[2] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

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

[4] Kenneth P Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, 1996. ISBN 0137195842.

[5] Kenneth P Birman and Thomas Joseph. Reliable communication in the presence of failure. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[6] Kenneth P Birman and Robbert van Renesse. Software for reliable networks. *Scientific American*, May 1996.

[7] Roberto Bisiani and Alessandro Forin. Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, 1988.

[8] Roy Friedman. Using virtual synchrony to develop efficient fault tolerant distributed shared memories. Technical Report TR95-1506, Department of Computer Science, Cornell University, March 1995.

[9] David Gelernter. *Mirror Worlds.* Oxford University Press, 1991.

[10] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors.* PhD thesis, Stanford University, December 1995.

[11] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. Technical report, Stanford University, December 1990.

[12] James Gosling and Henry McGilton. *The Java Language Environment White Paper.* Sun Microsystems Inc, May 1996.

[13] Andrew Grimshaw, Adam Ferrari, Frederick Knabe, and Marty Humphrey. Wide-area computing: Resource sharing on a large scale. *IEEE Computer*, 32(5):29–37, May 1999.

[14] Mark Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, January 1998.

[15] Iona Technologies Plc. *OrbixTalk White Paper*, 1998.

[16] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstation and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–132, January 1994.

[17] R Kordale, M Ahamad, and M Devarakonda. Object caching in a CORBA compliant system. *USENIX Computing Systems Journal*, 9(4):377–404, 1996.

[18] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[19] Microsoft Corp. *DCOM Technical Overview*, 1996. White paper.

[20] Microsoft Corp. *DCOM Architecture*, 1998. White paper.

[21] Jeff Nelson. *Programming Mobile Objects with Java.* John Wiley & Sons, 1999. ISBN 0-471-25406-1.

[22] Object Management Group. *Event Service Specification*, March 1995.

[23] Object Management Group. *Naming Service Specification*, March 1995.

[24] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, February 1998.

[25] ObjectSpace Inc. *ObjectSpace Voyager Core Package Technical Overview*, December 1997.

[26] ObjectSpace Inc. *ObjectSpace Voyager ORB 3.0 Developer Guide*, March 1999.

[27] Vijay S Pai, Parthasarathy Ranganathan, Sarita V Adve, and Tracy Harton. An evaluation of memory consistency models for shared-memory systems with ILP processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1996.

[28] Alan Pope. *The CORBA Reference Guide.* Addison-Wesley, 1998. ISBN 0-201-63386-8.

[29] Arno Puder and Kay Roemer. *MICO is CORBA: A CORBA 2.2 Compliant Implementation.* Morgan Kaufmann Publishers, September 1998. ISBN 3-93258-842-8.

[30] Phil Rosenzweig, Miriam Kadansky, and Steve Hanna. The Java reliable multicast service: A reliable multicast library. Technical Report SMLI TR-98-68, Sun Microsystems Inc, September 1998.

[31] Andrew S Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995. ISBN 0-13-219908-4.

[32] Hann Wei Toh, Mohammed Yakoob Siyal, and Hinny Pe Hin Kong. Design of a heterogeneous DSM system with distributed objects. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1817–1825, June 1999.

[33] Robbert van Renesse, Kenneth P Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4), April 1996.

[34] Alexey Vaysburd. *Building Reliable Interoperable Distributed Objects with The Maestro Tools*. PhD thesis, Cornell University, May 1998.

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

[36] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5), September 1992.