A Distributed Execution Environment

for Shared Java Objects

Nadia Erdogan, Yunus Emre Selcuk, Ozgur Sahingoz

Computer Engineering Department
Electrical-Electronics Engineering Faculty
Istanbul Technical University
80686, Ayazaga, Istanbul-TURKEY
e-mail: erdogan@cs.itu.edu.tr
selcukyu@itu.edu.tr
o.sahingoz@hho.edu.tr

Abstract

This paper discusses the implementation of a distributed execution environment, DJO, which supports the use of shared Java objects for parallel and distributed applications and provides the Java programmer with the illusion of a network-wide shared object space on loosely-coupled distributed systems. DJO supports shared objects through an implementation of multiple reader/single writer write-invalidate DSM protocol in software, providing the shared memory abstraction at object granularity. Object distribution and sharing are implemented through the replication mechanism, transparently to application. The system enforces mutual consistency among replicas of an object. The main benefits of DJO are enhanced availability and performance due to the replicated object model and easier application design, as the underlying software takes care of distribution and memory consistency issues.

Keywords: replication, consistency management, concurrency control, distributed object system, Java.

1 INTRODUCTION

In a distributed system, processes and users require access to shared data for meaningful cooperation. Traditional high performance computing often uses message passing to share information in parallel applications on distributed environments [22],[17]. With this approach, the developer has the advantage to control communication occurring in these applications and can adjust it to avoid unnecessary latency which can effect overall performance. This control, however, implies a responsibility to plan every communication detail. As a result, application development becomes extremely difficult and time consuming [10].

As an alternative to message passing, the *shared memory* paradigm offers an attractively simple programming model for application development. It has been extended to distributed memory machines where it is usually referred to as *distributed shared memory (DSM)* [18],[16]. DSM provides a *logical* single address space which is transparently partitioned over a set of physically distinct nodes. Programming with this model removes the need to explicitly consider communications, as this is handled by the DSM service, thus reducing application development complexity. Transparency is achieved by applying consistency models that take care of propagating changes to the shared information in the distributed environment.

This paper presents a distributed execution environment for shared Java objects (DJO) with its underlying software architecture [19], [20] . DJO is an effort to provide the distributed shared memory abstraction at object granularity. It supports the use of shared Java objects for parallel and distributed applications, thus providing the Java programmer with the illusion of a network-wide shared object space on loosely-coupled distributed systems.

During DJO design, we have adapted some of the techniques used in software based implementations of DSM systems to address the problems for sharing objects in a distributed object system. The design of DJO is based on the *replicated object model.* Replicated objects are local copies of logically shared Java objects. They allow for object methods to be invoked locally, eliminating the need to contact the remote object. DJO software architecture implements the multiple reader/single writer write-invalidate protocol to provide a memory consistency model which guarantees a consistent view of shared object state that matches programmer expectations.

DJO distributed execution environment consists of a collection of interconnected nodes, where each node executes a number of basic components that collectively form the runtime system. The runtime system is responsible for cashing replicas of objects and also for propagating invalidation requests and local updates of an object to the other replicas. An application interested in accessing a shared object contacts its local runtime system, which obtains a replica of the object and maps it into the application's address space. The application can then access the object through local method calls. A critical region encloses method invocations that modify the object state. Applications on other nodes can not access the replicas of the object until the control of the critical region is released. Entry and exit to a critical region is specified by special operations which alert the system, that, in turn, notifies all other nodes that maintain a replica of the shared object through invalidation messages on an entry and saves local changes on object state on an exit.

DJO is implemented in Java, and shared objects are a direct extension of Java objects. The main extension is that object state is distributed, which is implemented without modifying the Java runtime environment. The target application domain of the system includes all types of collaborative applications, including groupware and program development environments.

The layout of the body of the paper is as follows: Section 2 briefly discusses requirements and related work. Section 3 describes the read-replication algorithm on which the design of DJO is based, followed by the consistency model and coherency protocol that have been implemented. Section 4 presents the design of the software architecture. Section 5 provides detailed information on the user interface of the system. Section 6 presents some performance measures, and finally Section 7 concludes the paper with directions for future research.

2 REQUIREMENTS AND RELATED WORK

A distributed application allows an artibrary number of running components (programs) across any number of address spaces and nodes, cooperating as peers in implementing the application. Components of an *object-based* distributed application communicate and cooperate through *shared objects*. Shared objects are fine-grained, fully encapsulated passive entities that consist of private internal state and a set of operations by which that state can be accessed and modified. Each shared object is an instance of a class. Users of shared objects have several requirements:

- Location transparent access

- Efficient access
- High performance
- High availability
- Fault tolerance

Implementation of DJO makes active use of *replication* to address the performance and availability requirements mentioned above. The system transparently maintains replicas of shared Java objects in the address spaces of applications that access them. A replica encapsulates a local copy of the replicated object state and offers an interface to manipulate this state. Access to a replica is through *local* method invocations. Thus, by allowing a local rather than a remote copy to be accessed, replicas decrease access times, as delays in retreiving and updating the object are minimized. Furthermore, replication of shared objects also improves performance as concurrent accesses to replicas on several nodes become possible.

Replicas improve availability by making it possible for applications to progress even when one or more replicas become temporarily unavailable. Fault tolerance is achieved by ensuring that object data is kept consistent. Loss of any one replica does not result in updates being lost, if other replicas have copies of the same updates.

When a sytem concurrently maintains several copies of the same object, their consistency should be guaranteed. Different levels of consistency exist. Recent studies show that the overhead of coherence protocol messages limits the performance of a system considerably [11],[8]. Weak consistency protocols delay moving data and consistency information until absolutely required to do so by the memory model, thus reducing communication requirements significantly. DJO implements *release consistency*, which is a variant of weak consistency protocol, through the *write-invalidate* policy to lower the cost of coherence. DJO further tries to hide communication cost by using multithreading to overlap communication with computation.

Over the past decade, many software distributed shared memory systems such as Ivy [15], Midway [5], Munin [6] and TreadMarks [2] have been implemented on top of message passing hardware or on network of workstations. Early systems were page-based, just providing a linear memory of bytes and relied on the underlying memory management unit and operating system software for implementation. Recent advances in DSM systems are providing increasingly more support for the sharing of objects rather than portions of

memory. Several object systems that have been described in the literature provide facilities similar to those provided by DJO. Generally, they are based on new language/operating system solutions whereas DJO is implemented on Java Virtual Machine, thus providing a portable execution environment independent of any hardware/software. Some of these object systems are outlined below.

Munin [6] is a shared-variable DSM and offers the application developer a set of consistency maintenance algorithms. Each algorithm is targeted to a group of shared variables with specific characteristics. *Indigo* [14] is a software DSM system that is implemented on a PVM [22] platform. *CVM* [12] implements a shared memory system based on a user-level library, which provides the developer with primitives for synchronization and allocation of shared memory. The user can choose between a set of given consistency models. *Disom* [7] presents an object oriented framework for object sharing. Classes are made sharable by inheriting from particular super-classes, which define methods that facilitate the exchange of updates among interested nodes. The framework provides an implementation of entry consistency. *Mushroom* [13] supports a framework of replicated objects written in Java. Communication is based on events, and programmers use events to describe a change that effects the replicas of an object. *Orca* [4] supports object replication and migration with strong consistency guarantees. However, all objects in this system must be written in a special Orca language.

Based on the observation that large and complex systems cause overhead that offsets their benefits, we have tried to keep the design of DJO as simple as possible. Instead of supporting multiple consistency models, as many other systems do, a single consistency model, one that we think suits groupware applications the most, is provided. We aim at an execution environment that is easy to use and does not burden the application developer with unwanted/unneeded abstractions and functionality. However, if needed, more complex services could be built on top of those provided.

3 READ-REPLICATION ALGORITHM

Several algorithms have been developed for implementing the shared data model [10]. These algorithms can be classified according to the strategy they use to distribute the shared entities: *migration* or *replication.* Migration implies that only a single copy of an objects exists at any time, so the object should move to the requesting node for exclusive access. Replication, on the other hand, allows for multiple copies of the same object to reside in several address spaces. Replication reduces the cost of read operations that do not alter the

object state, since it is possible to simultaneously execute such operations locally on multiple nodes. However, operations that modify the state of the shared object become more expensive because its replicas have to be invalidated or updated to maintain consistency. If the ratio of reads over writes is large, this extra expense may be offset. DJO implements the ***Read-Replication : multiple readers/single writer*** [1] strategy to distribute shared objects. Object invocations are divided into two types: read accesses that do not change the state of the object and write accesses that modify the object. DJO allows for either:

- *multiple nodes* with read-only replicas of the shared object - the object is replicated on two or more nodes and each node has read access to its copy while none of the nodes have write access, or

- *one node* with a read/write replica - no two nodes may be modifying separate copies of an object at the same time.

On a request for a read access on a shared object that is currently not local, the system communicates with remote nodes to get a read-only replica of the object into the caller's address space, which may only be possible if no writable replica exists in the system. A request for a write access to a shared object that is either not local or for which the local node has no write permission may only proceed after all replicas at other nodes are invalidated, thus preventing them from being accessed. The read-replication algorithm is consistent because a request for a read access always returns a replica with an internal state that reflects the results of the most recent write access.

3.1 Consistency Maintenance

As stated in Section 2, replication improves performance by allowing concurrent access to replicas at multiple nodes. However, if the concurrent accesses are not controlled, they may be executed in an order different from that expected. A memory is **coherent** if the value returned by a read access is always the value that was expected. [1]. Thus, to maintain the coherence of shared objects, a mechanism that controls or synchronizes the accesses is necessary. A *consistency model* defines a specific kind of coherence provided by the system while a *coherency protocol* is responsible for managing object data so that the required level of consistency is actually provided.

3.1.1 The Consistency Model: Release Consistency

Consistency models define the order in which accesses to shared memory are seen by interested parties. A number of different models have been proposed in the literature such as *sequential consistency*, *causal consistency*, *PRAM consistency*, *weak consistency*, *release consistency*, *and entry consistency* [1]. Consistency models can be divided into two major categories: *strict models* and *relaxed models*. In general, the stronger the consistency level, the higher is the latency its implementation produces [3]. Strict consistency models order each access operation individually, while relaxed models combine a set of operations and impose an order on these sets. With strict consistency models, every write access results in an invalidation/update operation on all replicas. However, not all applications require to see all updates to a shared object, in which case relaxed consistency models perform better [3]. Relaxed models allow replicas to become inconsistent and perform coherence operations at specific user defined synchronization points within the program. The overall effect is reduced network traffic. One disadvantage, however, is that the programmer is expected to label programs with synchronization operations that act to separate conflicting sets of access operations.

DJO implements ***Release Consistency*** which is a variant of relaxed consistency model. [9] to maintain the coherence of shared objects. In a release consistent environment, object invocations that modify the object (write accesses) require synchronization. Two synchronization operations are defined to differentiate between entry and exit to critical regions which enclose write accesses to shared objects: an **acquire** operation tells the system that a critical region is about to be entered and a **release** operation indicates that a critical region has just been exited. DJO requires the programmer to determine, for every operation, if it modifies the object to which it applies, and to explicitly use these special operations on the target object to signify the beginning and end of a set of object invocation which result in modifications on object state. In the current implementation, each shared object that is to be write accessed is associated with a synchronization variable, actually a lock, to enforce concurrent accesses to happen sequentially. The system meets the following conditions, with the support of the underlying coherency protocol, to achieve release consistency.

- It does not allow an acquire access to perform until an up-to-date copy of the shared object associated with the lock is brought into the address space of the requesting application.

- It is ensured that no replica is accessible by any other application even in read access mode while an application is active in a critical region.
- After a release operation on an object has been completed, all read access requests of other applications are not allowed to proceed until they all receive an updated replica into their address space.

3.1.2 The Coherency Protocol

The coherency protocol is responsible for managing shared objects so that the conditions to provide release consistency are satisfied. The main issue is the synchronization of write accesses to objects in such a way as to insure no application reads old data once a write access has been completed on some replica of the object. There are two approaches: *write-update* and *write invalidate* [21]. Write-update broadcasts the effects of all write accesses to all nodes that have replicas of the shared object. This approach is usually considered to be expensive since a broadcast is needed on every write. In the write-invalidate scheme, on the other hand, *invalidations are sent* and *modifications are requested*. The basic concept is to send an invalidation message to all nodes that hold a replica before doing an update. Applications ask for updates as they need them.

DJO adopts the ***write-invalidate*** protocol and implements it in the following way:

- An object accessed in read-only mode can be replicated on multiple nodes.
- When an application requests a write access to an object through an acquire operation, a multicast message is sent to all nodes that have a read-only replica of the object to invalidate them, and all invalidations are acknowledged before acquire can proceed. This approach prevents the other applications from reading object data that is out of date.
- Any application that requests to read access an object is not allowed to if a writer to the object already exists.

With this scheme, updates are propagated only on requests for object access. Therefore, an invalidated replica on a certain node is updated only when it receives an access request. Meanwhile, several updates to the object may have taken place on other nodes before communication for state transfer becomes necessary. Furthermore, it should also be noted

that, several invalidated replicas with different state data may exist in the system at any time, but all valid replicas will hold the same object state data.

4 REPLICATED OBJECTS SOFTWARE ARCHITECTURE

Figure 1. shows the software architecture that implements the replicated shared object design discussed in the previous sections. A runtime system executes on each node and is responsible for replica management, and propagation of updates and invalidation requests. After completing initialization tasks, the system starts listening to a given address for incoming request messages and spawns a new thread to fulfill each request. Applications interested in shared objects contact their local runtime systems.

A main concern in the design has been to minimize communication overhead, which brings a major limitation on performance. We try to hide communication cost by using Java multithreading to overlap communication and computation whenever possible. All data structures and management routines are thread-safe.

The software is structured in layers. At the lowest level is the Communication Management Module. It is responsible for providing elementary communication mechanisms. Next, the Coherency Protocol layer provides routines to perform elementary actions such as transfering object state, invalidating replicas, etc. The Synchronization Module at the same level implements a locking mechanism to guarantee exclusive access to shared objects. Finally, at the highest layer, the Consistency Manager is responsible for implementing the consistency semantics.

4.1 Communication Management Module

The Communication Management Module provides local and remote communication support. Communication between system components takes place through messages that carry requests and resulting management information between local and remote threads. A single message format has been used for simplicity. Each message contains information about the request type, the unique object name, IP addresses identifying either the source or destination of the requesting node, and other fields containing management information required by the request type. Each request interprets those fields differently, extracting information relevant to its type. Communication is carried out through UDP sockets and RMI [23] calls. UDP sockets are used for internal messaging between system components. RMI calls are

used to contact remote nodes, for example, when transferring the state of an object. Multiple threads are created for communication via sockets. Threads resolve the received messages and take the necessary actions, which could be generating a reply message providing the information asked to another thread, or calling a remote method of a runtime system resident on another node. Threads use a particular port to listen to replies to their requests from other threads. Figure 2. shows the interactions in the communication module.

4.2 The Coherency Protocol

The coherency protocol is implemented by three management components: *memory management component, ownership management component*, and *distribution management component.*

4.2.1 Memory Management Component

Memory on each node in the system is effectively a part of the total shared object space and continues to exist as long as the node takes active part in the DJO environment. The Memory Management Component is responsible for the management of replicas that are resident in the shared address space of each node. It resolves a given object descriptor, its system-wide unique name, into a location in this address space and processes access operations. Each application has direct access to a *local memory region* which is private to the application (heap area), and an indirect access to *a shared memory region* which is the shared address space on the node where storage is allocated for object replicas. An invocation on a shared object requires the copying of the replica from the shared memory region into the local memory region, thus enabling *local method calls* on the object.

The shared memory region on a node is organized into two Java Hashtables, **ObjStateTable** and **ObjInfoTable**, that hold, respectively, current state and management information for the replicas that are located on that particular node. ObjStateTable treats objects as if they had no specific type, dealing with them as being of type Object, root class of all classes in Java. Thus, shared objects of any type can be put in the table. As the type information is lost, a cast to the correct form has to be performed while retrieving the object from the table.

A valid replica of an object is retrieved from the ObjStateTable on local or remote demand. Modification on state of a replica, either resulting from local invocations or transferred from a remote node, is stored in this table.

ObjInfoTable contains management information that is used to locate and transfer object state and to invalidate replicas. An entry in the table consists of the following fields:

name: Applications that share an object call it by the same name, which is unique across all replicated copies. The name of an object is a user defined character string. The hash tables ObjStateTable and ObjInfoTable are actually manipulated using unique object names as the keys.

probable owner: Each shared object in the system has a single owner. The owner of an object is a unique node which is either the creator of the object or holds the only writable copy of the object. The probable owner field either points to the true owner of an object or provides a hint through which the true owner of an object may be located.

copyset: Copyset is a set of nodes that hold valid replicas of an object. It is maintained by the owner node.

status: A replica of an object may be in one of the following states at any time.

invalid: an invalidated replica is present on the node.
readable: a valid replica is available and is not locked.
writable: a locked replica is available and its other replicas are invalidated.

Figure 3. shows the resulting state transitions of an object replica on different operations.

Memory Management Component also maintains a data structure, **ServerList**, which keeps information about the set of nodes that are currently participating the system. The ServerList carries addressing information that is needed to contact the run time systems resident on those nodes.

4.2.2 Ownership Management Component

The Ownership Management Component is responsible for the administration of the ownership of replicas. DJO adopts the dynamic-distributed scheme for ownership management [10]. The owner of an object is not fixed and moves around the system, hence

introducing the problem of locating it. To address this problem, the concept of *probable owner* has been used. The probable owner need not *own* an object but is responsible for tracking its current owner. Each node associates a probable owner with each replica. This information is just a hint. If the relevant field contains the address of the local node, then that node is the true owner of the object. Otherwise, it provides the beginning of a sequence of nodes through which the true owner may be located. Requests are forwarded until the true owner is reached. The probable owner field is updated whenever an up-to-date copy of the object is received. The main activity of this unit is to identify the location of the true owner of an object.

4.2.3 Distribution Management Component

Distribution Management Component has the functionality required to send and receive replicas of objects from remote nodes and is closely linked to the Communication Module. A transfer operation accepts an object's name and returns its current internal state and management information. It should be noted that, we assume the receiving application to already possess the required operation code for the object (its class declaration) and thus, transfer only object state. Otherwise, the class declarations would need to be dynamically loaded, too.

The Distribution Management Component is also responsible for sending out invalidation messages before a write access to an object is permitted to proceed. Copyset is passed along with the ownership property as a node becomes the new owner of an object. Invalidation messages are sent to all nodes on the copyset to prevent access to old data. The copyset at different nodes may be different. However, the owner node knows precisely the set of nodes that currently hold a replica of the object.

4.3 Synchronization Module

Most parallel applications need to use some kind of synchronization mechanism to order and control concurrent access to shared objects in cases where access results in modification of the object state. DJO provides locking primitives to achieve mutual exclusion. When an application possesses a lock on an object, it can be sure that it will be granted exclusive access to a recent copy of the object. Only the owner of an object can lock it. Therefore, ownership of the object has to be gained before applying for a lock. A lock is assigned to an application following its request to *acquire* an object and is set free on its next *release* request

on that object. It is the programmer's responsibility to take care so that an object does not remain locked forever.

4.4 Consistency Manager

The Consistency Manager implements the release consistency semantics described in the previous section, which includes access and synchronization operations on shared objects. The access operations are directed to the underlying coherency protocol. Synchronization operations are, in turn, mapped to corresponding operations provided by the synchronization module.

The Consistency Manager also supports *acquire* and *release* operations on shared objects. The acquire operation allows entry into a critical region by acquiring a lock from the synchronization module. The operation specifies the target object as the parameter of the acquire process and, with this information, the synchronization module can determine if it conflicts with another action. For example, some other application may already be holding a lock on that particular object. The release operation ends the critical region by notifying the coherence protocol to save the object state in the nodal replica. Local modifications made to the object copy in the local memory region are transmitted to the corresponding entry in the ObjStateTable. After the operation completes, the lock is released.

5 USER INTERFACE

Shared Java objects in the distributed environment become accessible regardless of their location through a small yet powerful set of calls that is supported by the underlying software architecture. They implement the release-consistent memory model, allowing new objects and replicas to be created, and replicas to be updated, invalidated or destroyed. They return either the desired resulting information or an error code. The user should check the return code to insure correct program flow.

The user interface consists of the calls listed in Table 1. The functionality of each is described in detail in the following sections and the Appendix section includes sample code that demonstrates their use in program context.

5.1 Create a Shared Object

```
Int djo.CreateObject(Object dsmobject, String ObjName)
```

Naming as well as shared memory space allocation is achieved with the CreateObject call. An application that wishes to introduce a new shared object uses this call *to register* it within the system. The caller provides the unique name of the object, a string, with the class data, in the parameter list. Any other application on a remote node may share access to the object if it knows this unique name (identity) of the object. The local runtime system consults local and remote directories to see if an object with the given identity already exists. If an object with the given name already exists, the call returns with a nonzero value. If not, an entry is allocated for the object in the ObjInfoTable, where management information is stored. The caller node gets the initial ownership of the object. Similarly, a new entry is allocated in the ObjStateTable, where a copy of the object state is stored. Thus any remote application that requests access to the object can have a replica of it retrieved from the ObjStateTable. A replica created on a node remains there until it is explicitly removed.

5.2 Remove a Shared Object

```
boolean djo.RemoveObject (String ObjName)
```

This call removes a local replica and discards all relevant information if it is not in locked state. No action is taken for its replicas on remote nodes. The call returns true on success and false on failure.

5.3 Access a Shared Object in Read-Only Mode

```
Object djo.ReadObject(String ObjName)
```

An application issues a ReadObject call before an attempt to read access an object. The ReadObject call brings a copy of the object into the local memory region of the caller so that local method invocations can be carried out. The runtime system returns the requested object if a valid replica in the unlocked state already exists on the shared memory region of the calling node. If not, one of the following cases will be true:

i) It may be the first time a node which is not the owner of an object is trying to access it. Therefore, the local node does not possess a replica yet.

ii) An invalid replica exists on the node. That is, an access to the object is requested after it has been invalidated due to coherence actions. This means that changes have been made to the state of the object, and these changes should be transmitted to the local replica before it may be accessed again. As a replica already exists, the probable owner may be contacted.

Case i: A query message is broadcast to all remote nodes on the ServerList, asking for the object. Nodes that hold a valid replica respond to this message. No reply received implies an access error to a nonexistent object, in which case the call returns null. Otherwise, the first node that responds to the broadcast message is contacted. This node is either the true owner of the target object or, is not the owner but holds a valid copy. In the first case, if the owner has restricted access to the object by locking it, null is returned. Otherwise, the owner node adds the identity of the caller to the copyset directly. In the second case, the call initiates an operation that locates the true owner and has the identity of the caller added to its copyset. Next, object state and management information are transferred from the owner node to the shared memory region of the caller node. The call proceeds to update the probable owner field with the current owner of the object, changes object state into *readable* and returns the object to the caller.

Case ii: Starting with the probable owner field of the object, a sequence of nodes is scanned until a node which is either the true owner of the object or holds a valid copy is located. Next, the call proceeds following the steps described in case i.

After the ReadObject call returns with success, the caller receives a copy of the object into its local memory region and may issue local method invocations, which are expected to access the object state in read-only mode as changes in object state will have no global effect. Readers are expected to contact the system by issuing the read-object call before each access (or, after some user-definable period of time) to find out if they still hold a valid copy of the object and, if not, to have it updated.

5.4 Receive Exclusive Access to the Shared Object

```
Object djo.AcquireObject(String ObjName)
```

Through the AcquireObject call, an application gets permission to enter a critical region . If the call returns with success, the caller obtains both the most recent version of the object copy into its local memory region and a lock on the object that ensures exclusive access.

From then on, the caller may issue several updating method calls on the object. The call proceeds to consult the true owner of the target object to ask for its ownership if the caller is not already its owner. The current owner rejects the request if it already holds a lock on the object. If this is not the case, the current owner puts the object into invalid state and removes itself from the copyset. This eliminates the invalidation it would have otherwise received from the new owner of the object. It hands over ownership after updating the probable owner field of the object with the identity of the requesting node and returns object management information along with object state data, which is requested only in the case when a valid replica is not present on the caller's address space. Upon receiving the ownership of the target object, the new owner node locks the object, putting it into *writable* state. However, all valid replicas of the object need to be invalidated before allowing access to the object to guarantee that no conflicting copies exist. Invalidation messages are sent to all nodes on the copyset of the object. The call waits for invalidations to be acknowledged to prevent the existence of old copies. After the invalidation process completes, the call purges the copyset as those nodes no longer possess a valid replica, inserts the identity of the local node into the copyset of the object and returns the object to the caller, which can then proceed into its critical region. Now, method invocations that result in changes in the object state may be issued. It should be noted that these modifications take place on the copy of the object located in the local memory region of the application and are not transmitted to the replica that resides on the shared memory region of the node (and therefore will have no global effect) until the critical region is exited through a ReleaseObject call.

5.5 To Release Exclusive Access and Save Modified Object State

```
boolean djo.ReleaseObject(Object Dsmobject,String ObjName)
```

ReleaseObject call ends activity in a critical region after transmitting local modifications of object state to the only replica of the object on the shared memory region. The call proceeds to verify that the caller is the true owner of the object and returns error if there is a conflict. Next, the entry allocated to the object in the ObjStateTable is replaced with the new state data provided by the caller. Finally, as the object is unlocked, the status field in the ObjInfoTable is updated to readable to reflect the current state.

Normally, a lock is released upon an application's request to exit a critical region. Therefore, the application should take special care so that the object would not remain locked forever. The system has no timeout mechanism to automatically release locks.

6 PERFORMANCE EVALUATION

We have performed simple tests to evaluate the performance of DJO system on a LAN environment of five INTEL architecture machines (Pentium II 350 MHz with 128 MB RAM) connected through a 10 Mbps Ethernet and running Windows 98 SE. JDK 1.3 was used in the testing environment. The data was obtained by subtracting the values returned by System.currentTimeMillis() method, which was invoked just before and right after DJO calls. The tests were repeated 10 times for 15 different objects and the arithmetic average of the measurements are reported. Table 2. presents the resulting performance figures, in milliseconds.

The tests included a simple application that involved concurrent access, either read or write, to shared Java objects by all participating nodes. Objects had a size of approximately 100 bytes. Each node acquired mutual exclusion before a write-access to an object and released it after completion. The measured figures account for the times spent in system calls, as described below.

Initialization: time taken to initialize a local runtime system before a node participates the DJO environment. Most of the measured time is consumed by the initiation of the Java "rmiregistery" process.

Creating new object: time taken to create an object with a unique name and register it within the system.

Read-only access: time taken to retrieve a i) locally available or ii) remotely available valid replica into the address space of the caller process.

Receive exclusive-access: The total time taken to receive a valid replica in locked state after obtaining ownership of object. It includes the time spent to send invalidation messages to 5 valid object replicas and waiting for arrival of acknowledgement messages.

Release exclusive-access: time taken to transmit local object state to nodal replica and to unlock object.

The results presented suggest that DJO can perform well. They were achieved without any specific optimization and it should be possible to make significant improvements on them.

We further plan to run the system on a larger number of nodes to test it for scalability and to observe effects on its performance.

7 CONCLUSION AND FUTURE WORK

This paper presents the design and implementation of a distributed execution environment for shared Java objects. Object sharing is implemented through the replication mechanism. Each application requesting access to a shared object receives a copy into its local address space. Object access is through invocation of methods provided by the object interface. The system enforces mutual consistency among replicas of an object transparently. A prototype implementation of the system has been completed.

DJO distributed execution environment has several advantages. One main advantage of the system is that it simplifies distributed application design. Programmers may concentrate on the application logic and receive the benefits of a distributed execution environment through a simple yet powerful user interface, as issues related to distribution, access and consistency of shared objects are handled transparently by the underlying system. Programmers do not need to write any extra code for object sharing and distribution.

The replication strategy employed in the distribution of shared objects not only decreases access times and increases parallelism, thus resulting in better application performance, but also makes the implementation of fault tolerant system possible.

Another advantage of the system is its being based on JVM, which ensures portability across a variety of hardware and software platforms .

DJO design can be enhanced with certain features. The current implementation does not employ a memory management algorithm for replicas on a node. Since the size of the shared memory region on a node is physically bounded, only a limited number of replicas can be held. The system rejects requests when this limit is reached. However, a better approach would be to remove replicas to free some memory space. As future work, we intend to study memory replacement algorithms, concentrating on locating objects that are particularly suitable for removal.

Another feature which we wish to include in our work is extending shared objects with support for security and persistence. The creator of the object can specify a security policy

during registration and the system can control access accordingly. Providing support for persistent shared objects would also ease code development significantly.

8 REFERENCES

[1] S.V. Adve, K. Gharachorloo, Shared Memory Consistency Models: A Tutorial, IEEE Computer, 29(12), (1996) 66-76.

[2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H.Lu, R. Rajamony, W. Yu, W. Zwaenepoel, TreadMarks: Shared Memory Computing on Networks of Workstations, IEEE Computer, 29(2), (1996) 18-28.

[3] H. Attiya, J. Welch, Sequential Consistency versus Linearizability, ACM Trans. on Computer Systems, 12(2), (1994) 91-122.

[4] H. Bal, F. Kaashoek, A. Tanenbaum , Orca: a Language for Parallel Programming of Distributed Systems, IEEE Trans. on Software Eng.,18(3), (1992) 190-205.

[5] B. Bershad, M. Zekauskas, W. Sawdon, The Midway Distributed Shared Memory System, Proc. 38th IEEE Int'l Compcon Conf., (1993) 528-537.

[6] J. Carter, J.Bennet, W. Zwaenepoel, Implementation and Performance of Munin, Proc.13th ACM Symp. on Operating Systems Principles, (1991) 152-164.

[7] M. Castro, P.Guedes, M. Sequeira, M.Costa, Efficient and Flexible Object Sharing, Proc. 1996 Int'l Conf. On Parallel Processing (ICPP'96) (1), (1996) 128-137.

[8] A. L. Cox, P. Keleher, H.Lu, R. Rajamony, W. Zwaenepoel, Software versus Hardware Shared Memory Implementation, Proc. 21st Annual Symp. On Computer Architecture, (1994) 106-117.

[9] S.Dwarkadas S, P. Keleher, A.L. Cox, W. Zwaenepoel, Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology, Proc. 20th Ann. Symp.on Computer Architecture, (1993) 144-155.

[10] A. Judge, P.A. Nixon, V.J.Cahill, B. Tangney, S. Weber, Overview of Distributed Shared Memory, Tech. Report TCD-CS-1998-24, http://www.cs.tcd.ie/publications/tech-reports/

[11] P. Keleher, A. L. Cox, W. Zwaenepoel, Lazy Consistency for Software Distributed Shared Memory, Proc. 19th Annual Symp. On Computer Architecture, (1992) 13-21.

[12] P. Keleher, C-W. Tseng, Enhancing Software DSM's for Compiler Parallelized Applications, Proc. 11th Int'l Parallel Processing Symp. (IPPS'97) (1997).

[13] T. Kindberg, G. Coulouris, J. Dollimore, J.Heikkinen, Sharing Objects over the Internet: the Mushroom Approach, Proc. IEEE Global Internet, (1996) 67-71.

[14] P. Kohli, M. Ahamad, K. Schwan, Indigo: User-level Support for Building Distributed Shared Abstractions, Concurrency:Practice and Experience, 8(10), (1998) 1-29.

[15] K.Li, Ivy: A Shared Virtual Memory System for Parallel Computing, Proc. of ICPP'88, 2, (1988) 94-101.

[16] K. Li, P. Hudak, Memory Coherence in Shared Virtual Memory Systems, ACM Trans. On Computer Systems, 7(4), (1989) 321-359.

[17] Message Passing Interface Forum, http://www.mpi-forum.org/

[18] L. Nitzberg, V. Lo, Distributed Shared Memory: A Survey of Issues, IEEE Computer, 24(8), (1991) 52-60.

[19] O. K. Sahingoz, Implementation of a DSM System Based on Read Replication Algorithm, Ms.C. Thesis, Istanbul Technical University, 1998.

[20] Y. E. Selcuk, Implementation of a Distributed Shared Memory System, Ms.C. Thesis, Istanbul Technical University, 2000.

[21] P. Stenstrom, A Survey of Cash Coherence Schemes for Multiprocessors, IEEE Computer, 23(6), (1990) 12-24.

[22] V. S. Sunderam, PVM: A Framework for Parallel Distributed Computing, Concurrency: Practice and Experience, 2(4), (1990) 315-339.

[23] Sun Microsystems, Java Remote Method Invocation Specification. Technical Report, Sun microsystems, Mountain Vıew, CA, USA, 1996.

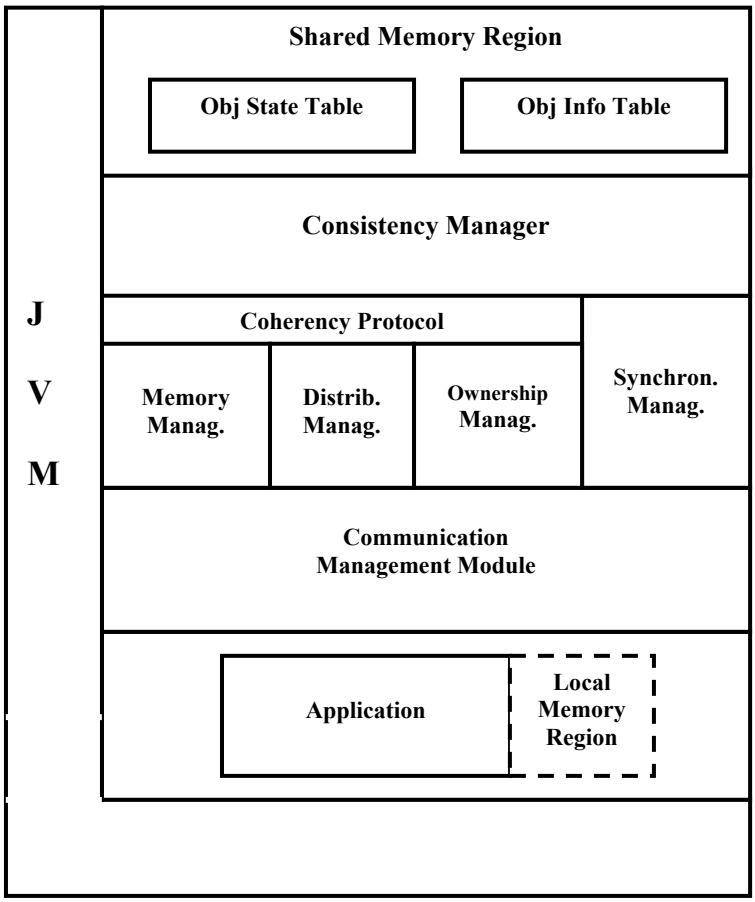
Shared Memory Region
Obj State Table
Obj Info Table
Consistency Manager
J
V
M
Coherency Protocol
Memory Manag.
Distrib. Manag.
Ownership Manag.
Synchron. Manag.
Communication Management Module
Application
Local Memory Region

Figure 1.

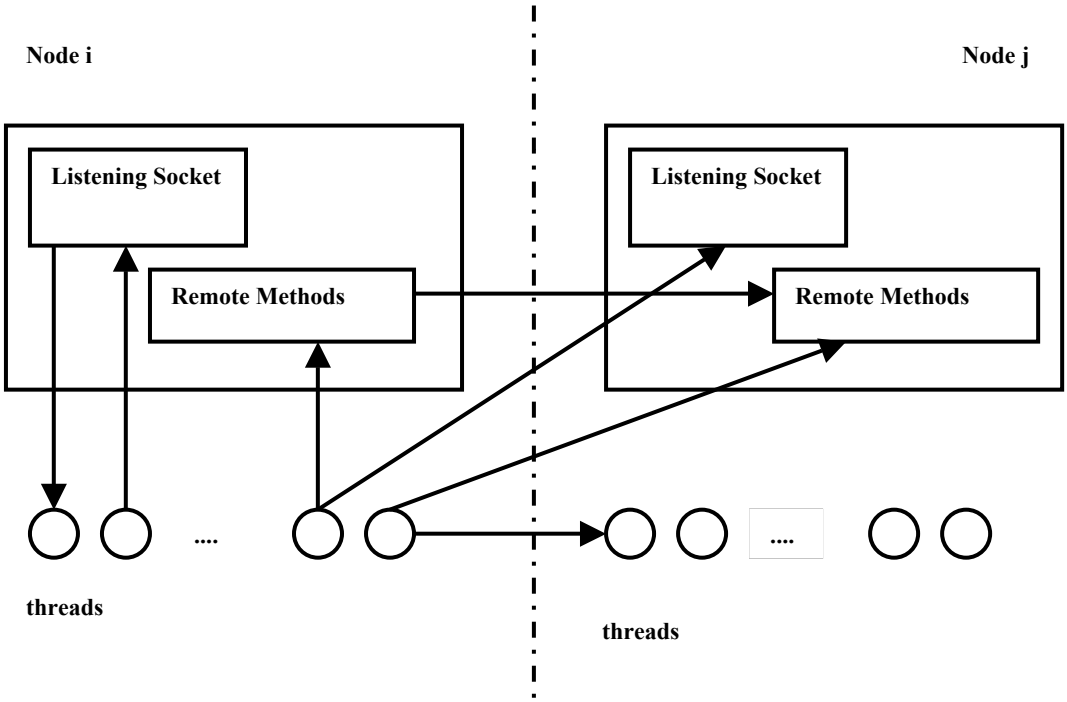
Node i
Node j
Listening Socket
Remote Methods
Listening Socket
Remote Methods
....
....
threads
threads

Figure 2.

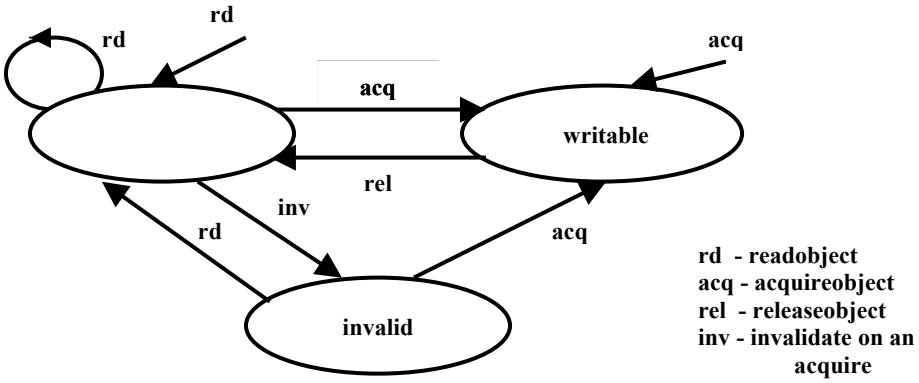
rd
rd
acq
acq
writable
rel
inv
rd
acq
invalid
rd - readobject
acq - acquireobject
rel - releaseobject
inv - invalidate on an
acquire

Figure 3.

Table 1.

Interface Call	Semantics
Int djo.CreateObject(*Objectdsmobject, String ObjName*)	Registers a new shared object
boolean djo.RemoveObject (*String ObjName*)	Removes local replica
Object djo.ReadObject (*String ObjName*)	Creates a local replica and permits read-only access
Object djo.AcquireObject (*String ObjName*)	Provides a locked local replica and permits caller to proceed into its critical region
boolean djo.ReleaseObject (*Object dsmobject, String objName*)	Updates local replica and releases the lock

Table 2.

Operation	Elapsed Time (ms)
Initialization	2160
Creating new object	30
Read-only access i) local replica ii) remote replica	 17 38
Receive exclusive-access	215
Release exclusive-access	84

APPENDIX

Class declaration for shared Java object penguin:penguin.java

```
public class penguin implements java.io.Serializable
{
  private int    age;
  public  int    height;
  private String nickname;
  public  String name;

  public penguin()
  {
    age = 0;
    height = 20;
  }
  public void setAge( int i )
  {
    age = i;
  }
  public int getAge( )
  {
    return age;
  }
    public void setHeight( int i )
  {
    height = i;
  }
  public int getHeight( )
  {
    return height;  }
}
```

Sample code that creates a shared object named "pepe":

```
.....
pepe = new  penguin( ) ;
pepe.setAge ( 1 ) ;
```

```
pepe.set.Height ( 25 ) ;
try
{
   status = djo.CreateObject ( pepe,  "pepe" ) ;
}
catch (Exception e ) { .... }
if ( status != 0 ) { ..... error....
....
```

Sample code to read access the shared object "pepe" on a node i:

```
....
int age;
int height;
…
pepe1 = new  penguin( ) ;
pepe1 = (penguin) djo.ReadObject ( "pepe");
age = pepe1.getAge ( );
height = pepe1.getHeight ( );
.....
```

Sample code to read/write access the shared object "pepe" in a critical section on a node j:

```
…..
boolean status;
pepe2 = new  penguin( ) ;
pepe2 = (penguin) djo.AcquireObject ( "pepe");
    pepe2.setAge ( 3 );
    pepe2.setHeight ( 40 );
 status = (penguin) djo.ReleaseObject ( pepe2, "pepe");
...........
```

Captions of figures and tables

Figure 1. Overview of replicated object software architecture

Figure 2. Interactions in the Communication Management Module

Figure 3. State transition of an object replica

Table 1. The User Interface

Table 2. Measured Performance Figures of DJO