J-DSM: Sharing Java Objects in a Distributed Environment

Yunus E. SELCUK and Nadia ERDOGAN
Istanbul Technical University
Computer Engineering Department
Ayazaga, 80626, Istanbul, TURKEY
e-mail: yeselcuk@papirus.library.itu.edu.tr
erdogan@cs.itu.edu.tr

Abstract: This paper presents the design and implementation issues for the use of DSM techniques to allow shared access to objects in an object-oriented distributed environment. J-DSM supports sharing of Java objects, thus allowing a finer control over the granularity of sharing. J-DSM elaborates on two main classes of shared objects, *mobile and stationary*, which can be accessed through DSM and RMI mechanisms, respectively. Shared objects are uniquely named and manipulated through the J-DSM interface.

Key-words: distributed shared memory, dsm algorithms, consistency and coherence.

1. INTRODUCTION

Loosely-coupled distributed systems have evolved using message passing as the main paradigm for sharing information. Other paradigms used in loosely-coupled distributed systems, such as remote procedure call (RPC) or remote method invocation (RMI), are usually implemented on top of an underlying message-passing system. On the other hand, in tightly coupled architectures, the paradigm is usually based on shared memory for its simple programming model. The shared-memory paradigm has recently been extended for use in more loosely-coupled architectures and is known as *distributed* shared memory (DSM). The DSM abstraction gives these systems the illusion of physically shared memory and allows the programmer to use the shared-memory paradigm [1].

DSM systems have many advantages over message-passing systems [1,2]. In DSM systems, processes share data transparently across node boundaries; data faulting, location and movement are handled by the DSM system. Thus, they provide the user a simple shared memory abstraction automatically, while message-passing systems require data movement to be specified by the programmer.

Object oriented systems have usually relied on remote procedure calls as the fundamental method for accessing objects in distributed environments. This model prevents simultaneous accesses to copies of an object on multiple nodes since all accesses to an object are sent to a single node. Another advantage DSM gives over RPC is the ability to transparently migrate and replicate data so that it can be accessed on multiple nodes simultaneously.

Most research done on DSM systems has concentrated on page-oriented shared memory. However, recent advances in DSM systems are providing increasing support for the sharing of objects rather than portions of memory [3]. This paper presents the design and implementation issues for the use of DSM techniques to allow shared access to *objects* in an object oriented distributed environment [4,5,6]. Our intention is to provide a DSM interface, J-DSM, which can be used to develop a wide range of applications in a portable and consistent manner, across a variety of hardware platforms. The flexibility of a DSM system relies on the granularity of the shared information and the independence from certain hardware. Much previous work on applying DSM techniques in an object oriented environment has concentrated on heavy-weight objects [7,8]. However, J-DSM supports sharing of smaller objects, namely Java objects, thus allowing a finer control over the granularity of sharing at the object level.

Taking a purely DSM-based approach can be too limited in a flexible environment. For example, a remote object invocation mechanism is still required to support objects that have a fixed location such as those controlling physical devices. We have adopted a hybrid approach in J-DSM, where both DSM and RMI mechanisms are supported for object access, as function shipping and data shipping models of communication may be needed to develop efficient applications. J-DSM elaborates on two main classes of shared objects (Java objects): *mobile shared-objects (mso)* use DSM mechanisms for access - data is moved to the user transparently

through replication. *Stationary shared-objects (sso)*, on the other hand, are remote objects and rely on RMI mechanisms for method invocation. However, J-DSM hides all invocation details from the user.

Many approaches have been proposed to implement DSM systems [7,8,9,10,11]. Generally, DSM implementations are based on variations of write-update and/or write-invalidate protocols. Recent implementations use relaxed memory consistency models such as release consistency [8]. J-DSM uses the *read-replication* (multiple reader/single writer) strategy of distributing shared objects across the system. The replication of shared objects complicates issues in memory coherence. In general, applying unnecessary coherence operations can waste bandwidth, create extra CPU overhead and cause unnecessary access faults. We use the *release consistency* model with *write-invalidate coherence policy* where the owner of a shared data object is directly responsible for enforcing coherence.

J-DSM is fully implemented in Java [12] and is portable to any platform that supports JVM (Java Virtual Machine). It is tested on a network of NT 4.0 workstation/server and Win9x platforms of Intel processors.

2. DESIGN OUTLINE

The current implementation consists of a set of abstract Java classes that implement both mobile and stationary shared-objects. A list of the important classes with their functionality is given in Fig. 1.

DSMImpl	the server class that imple ments J-DSM on a node
DSMInfo	contains management info for each shared object
DSMExecutor	interprets request messages and takes approp.action
DSMPacketData	implements message format
DSMSpawnerInfo	keeps RMI management info for stationary objects
DSMSpawnerThread	responsible of stationary objects

Figure 1. Class architecture of J-DSM

A request to access a mso results in the replication of a local copy on the node where the request originated. Sections 3 and 4 give details on the implementation issues of memory management, distribution, ownership and the coherence policy.

Stationary shared-objects **do not need** replication. They are remote objects and are accessed through remote invocations to their methods via RMI. However, several sso's of the same class can be created with the same or different names on different nodes if their existence is required for the efficient solution of a distributed problem. A *Spawner Object* is responsible for the maintenance of all stationary shared-objects of a class. A class a sso belongs to is called a stationary DSM class and is derived from an ordinary user class. The methods which are desired to be invoked remotely are declared in a Java interface. Calls to these methods are invoked from the corresponding DSM class after passing through certain conditional blocks which determine the current status of the object. Certain remote methods (see Section 6.2) are also added for management purposes. These remote methods are the same for all user classes except for the classname in typecasting operations. This approach is similar to the *Activation* facility which comes with JDK 1.2. J-DSM includes a preprocessor utility which creates the source code of the necessary spawner and stationary DSM classes, thus hiding all remote invocation details from the user.

2.1 Information Structure

We assume an implementation where a shared object directory is distributed among nodes that participate the J-DSM system. An entry of the directory contains management information for each object, local or replicated, to be used to locate and transfer objects and to invalidate replicas. The management information of a shared object has the following fields:

Name: Two processes in an application share an object if they call it by the same name. Therefore within each application, all shared objects must be named uniquely across all of their replicated copies. The item's name is its unique identifier. Name is not stored in a String member of the DSM management information class, but it is actually used for indexing of the hashtable where this information is stored.

Owner: The identity of the unique owner node which owns the only writable copy of the shared object and has the right to update that object. A process should first get the ownership of data before it can write to it.

Copyset: Set of nodes that have copies of a shared object. This list is maintained by the owner node.

Probable Owner: Each node keeps track of the probable owner of each shared object. This information may be incorrect but it provides the beginning of a sequence of nodes through which the true owner may be located. When a copy of an object is needed, a request is sent to the probable owner. If the probable owner does not have a copy of the object, it forwards the request to its probable owner. The request is thus "forwarded" until the true owner, the node that has a copy of the object is reached.

Status: A shared object may be in one of the following states at any time:

readable: the data object is available and not locked

writable: the data object is available and its replicas are invalidated

available: The data block is present and contains valid data

locked: access to the object, except from the owner, is denied

Node List: Set of nodes currently participating the DSM system. Node list is actually stored in each DSM server and not in an array member of the DSM information class.

Spawner Table : This is a hashtable where the lookup name of a spawner is stored using the class name of a stationary shared-object as the index.

3. THE DSM ALGORITHM: Read-Replication

The algorithms for implementing DSM deal with the problem of distributing shared objects across the system.Two frequently used strategies are *migration* and *replication*. Migration implies that only a single copy of an object exists at any time, so the object should be moved to the requesting node for exclusive use. This strategy is preferred when sequential patterns of write sharing is prevalent. Replication, on the other hand, allows for multiple copies of the same object to reside in memories of different nodes. As we consider read sharing to be the characteristic of memory references in typical distributed applications, we have chosen to implement the *read-replication (multiple reader single writer)* [7] strategy to enable simultaneous accesses by different nodes to the same data and to minimize access latency.

With the read-replication algorithm, a read request results in fetching and creation of a replica of an object from a remote location to the caller's memory space. Thus simultaneous local execution of read operations at multiple nodes is possible. Only one node at a time can receive permission to update a replicated copy of a shared object. A write to a writable copy requires the use of other replicated copies be prevented. Therefore, we implement an *invalidation* based algorithm.

For each particular data object, the identity of the probable owner is kept. All requests go the probable owner, which usually is also the real owner. However, if the probable owner is not the real one, the algorithm forwards the request to the node representing the probable owner according to the management information kept in its DSM directory. For every read, write and invalidate request the probable owner field changes accordingly, to decrease to number of messages to locate the real owner.

The owner of an object also keeps its copy set, the list of nodes that have replicas of the shared object. The copy set goes together with the object to the new owner, which is also responsible for invalidations. For a write request, invalidation messages are sent to all nodes on the copy set.

4. CONSISTENCY AND COHERENCE POLICY

The replication of shared objects complicates issues in enforcing memory coherence. The coherence policy determines whether the existing copies being written to at one node will be *updated* or *invalidated* on the other nodes. A simple implementation of a *write-update* protocol, where a write updates all replicas of a shared object is likely to be inefficient, because many replicas may be updated, even if some of them are not going to be accessed in the near future. Therefore, we use the *write-invalidate* protocol for *release consistency*.

Release consistence [11] is a *relaxed* memory consistency model that permits a node to delay making its changes to shared object visible to other processes until certain synchronisation accesses occur. That is, it allows views of shared objects by different nodes to become inconsistent until subsequent synchronization events. Thus, release consistency results in better performance by letting write accesses to be pipelined and guarantees results equivalent to sequential consistency for a program that contains enough synchronization to avoid data races.

The write-invalidate protocol allows for many replicas of a "read-only" shared object, but only one copy of a "writable" object. All replicas of the shared object except one are invalidated before a write request can proceed. Our implementation of the protocol is as the following:

***For a read request*:** If the object is available, it is returned immediately. If not, a read request is sent to the probable owner and a copy is returned. The copy remains valid until an invalidation request is received.

***For a write request*:** If the object is writable, the request is satisfied immediately. Otherwise, a request for a copy of the object, along with a request for invalidation need to be sent such that the local copy becomes valid and writable, and the original write request may complete.

5. J-DSM IMPLEMENTATION

Management responsibility shared objects is distributed to all nodes on the system. The DSM algorithm is implemented by server processes present on each node. Communication between DSM components takes place through messages. Messages carry requests and resulting management information between local and remote

threads. We have chosen to use a single message format for simplicity and efficiency. Each message contains information about the request type, the address of the originator of the request, and various management data about the shared object. Subfields of the message are evaluated differently, depending on the request type.

All the communication is implemented by using UDP sockets and RMI calls. Remote methods are used whenever the state of an object is to be transferred. Otherwise, UDP sockets are used for internal messaging between DSM system's components. Multithreading is used whenever possible when communicating via sockets. Figure 2. depicts the interactions between several components on a node. The functionality of each component is as the following.

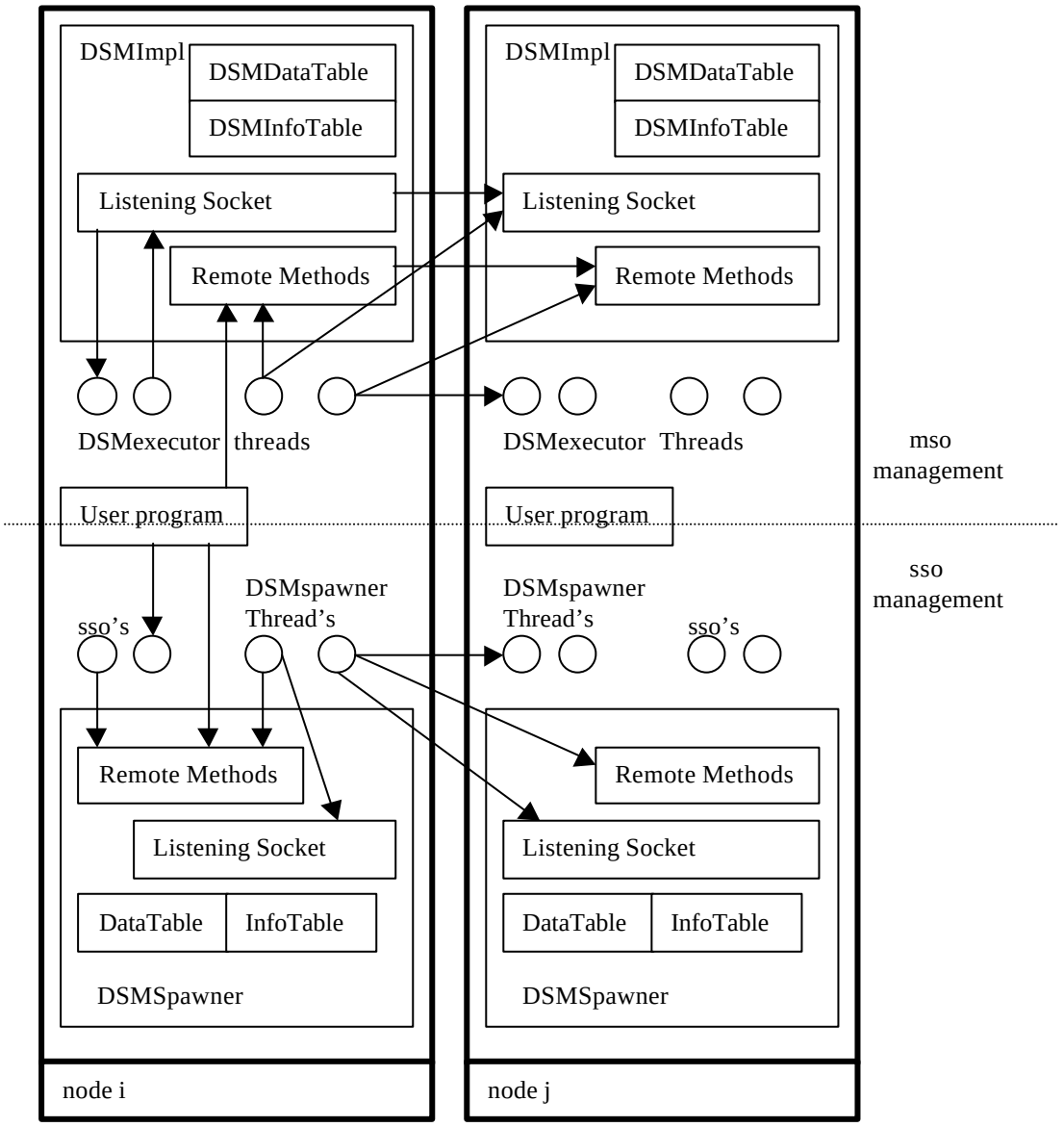

Figure 2. Interactions between system components

DSM Server, DSMImpl, is a multithreaded process which is responsible of maintenance actions for mobile shared-objects. Management information and current state of mso are stored in two hashtables named DSMDataTable and DSMInfoTable, respectively, both indexed with the unique names of shared objects. After completing the necessary initial tasks, DSM server process starts listening port 5000 for incoming request messages. A new DSM thread is spawned to fulfill each request.

DSM Threads resolve the received messages and take the necessary actions, like replying to another DSM thread with resulting information or calling a remote method of a DSM server resident on another node. Threads use port 5001 to listen to replies to their requests from other DSM threads.

Remote Methods A DSM server supports a set of remote methods which are either called by other DSM servers on different nodes or construct the user interface, which is explained in detail in Section 6. Some of these remote methods may spawn threads for selective multicast of a DSM message to the system.

DSM Spawner is a simplified version of a DSM server, as a stationary shared-object needs less management work than its mobile counterpart. Management information and current state of a sso is again stored in two hashtables indexed by the object's names, namely DataTable and InfoTable. However, DataTable is only accessed before an object is being recreated or after it has been finalized. A sso is finalized by JAVA's distributed garbage collector when all remote references to this object are finalized. Similar to the DSM server, the spawner always listens to port 5002 to receive management messages and creates a new thread to handle the request.

Spawner Threads are similar to DSM threads, except they execute fewer management commands. Spawner threads can communicate with other spawners of the same class and other spawner threads. Moreover, remote methods of the spawner can also be called when necessary. Threads use port 5003 for listening to replies to their requests from other DSM threads.

6. J-DSM USER INTERFACE

Several remote methods constitute the user interface of J-DSM. As mobile and stationary shared-objects are accessed differently, methods that manipulate them are implemented by different classes. Methods related to mobile shared-objecsts are implemented by the DSMServer class, while those related to stationary shared-objects are implemented by the DSMSpawner class.

6.1 Mobile shared-objects

Create : int DSMserver.CreateObject(Object dsmobject, String objName)

User processes use this method to create a mso. The class data and the unique name of the object to be created are supplied through the parameters. Local and remote directories are consulted to see if an object with the given name already exists. If not, an entry in the directory is allocated for the object and its ownership is assigned to the requesting process. Otherwise, the method returns with a nonzero value.

Remove : boolean DSMserver.RemoveObject(String objName)

A local mso is removed from the local node without considering its status. No action is taken for its replicas on other nodes. This function returns true on success and false on failure.

Read : Object DSMserver.ReadObject(String objName)

User processes call this remote method to read the current value of a mso. If a replica of the object doesn't exist on the local node, the entire system is queried for the true owner of the object. This method returns a replica of the object for use of the caller process, such that its methods can then be executed locally.

Read and Own : Object DSMserver.ReadLockObject(String objName)

The unique owner process of a mso has the right to update it and owns the only writable copy in its memory space. Therefore, a process should first get the ownership of an object before it can issue a write request. This call returns a valid copy of the object together with its ownership so that the caller gets permission to update it. The message contains an *invalidation request* such that all replicas except the local copy are invalidated by passing into the unavailable state. This method is implemented simply by executing the lock statement described below and returning a replica of the object after completion.

Write : int DSMserver.ModifyObject(Object dsmobject, String objName)

User processes call this remote method to update the value of a mobile shared-object. This method doesn't make any attempts to take the ownership of the object since the object should be owned first, as described above, before any attempt to modify it. If this is not the case, the method returns with an access error.

Lock/Unlock : Object DSMserver.LockObject(String objName, boolean lock, block)

This method is used to change the state of a mso. The probable owner of the object is searched for within the DSM system with selective multicast messages sent to each node on the nodelist of the object, each by a new thread. This method brings the most overhead to the DSM system. The object is locked if the lock parameter is true, unlocked otherwise. If the object is to be locked and the block parameter is true, the object is put in the unavailable state.

6.2 Stationary shared-object

create : boolean DSMspawner.CreateOtonomus(Object dsmObject, String dsmName, boolean force)
boolean DSMspawner.CreateOtonomus(String dsmName)

The first method for creating a sso is similar to that of mso, except for the addition of a boolean parameter. If the third parameter is true, an object is created even if another object already exists in the system. Replicas of a sso can only be created in this way. The second method is for creating an object with its default constructor. Both functions return true on success and false on failure.

own/lock : boolean DSMobject.DSMownOtonomus(boolean block)

This is one of the three remote methods which a sso should implement. The *invalidation request* is handled if the block parameter is true. Otherwise, the method proceeds the way its counterpart does for a mso.The second remote method that should be implemented by a sso, namely **void transfer(DSMobject pe, DSMInfo inf)** ,is used within DSMownOtonomus method. This reveales why the locking mechanism is embedded in objects – typecasting process is required here and the class of this particular data object is *a priori* knowledge.

lock : boolean DSMobject.DSMlockOtonomus(boolean lock, boolean block)

The functionality of this method is similar to that of mso.

Read/write:As sso is not necessarily replicated, their methods are invoked remotely. Thus reading and updating members of a sso is possible only through remote calls. The user needs to add read/write methods to the sso class if these actions are required by the application.

7. RESULTS

This paper presents a flexible and portable Java framework, J-DSM, for distributed shared objects. The main feature of J-DSM is its support for both mobile and stationary shared objects., which improves the flexibility of the sytem. Users can choose between the two access protocols to shared objects to suit the application's semantics for more efficient implementations.

A prototype implementation in Java has been completed, and initial results are encouraging. Future plans include work for improvements in performance and utilization of the framework for development of distributed applications.

References:

[1] B.Nitzberg and V.Lo, "Distributed Shared Memory:Asurvey of Issues and Algorithms", IEEE Computer, Vol.24, pp.52-60, Aug. 1991.

[2] M.Stumm and S.Zhou,"Algorithms Implementing Shared Memory", IEEE Computer, pp.54-64, May 1990.

[3]H.Bal, Programming Distributed Systems, Silicon Press, Prentice Hall, 1990.

[4] O. K.Sahingoz, “Implementation of a DSM System Based on Read Replication Algorithm”, Institute of Science and Technology, Ms.C. Thesis, 1998

[5] Y.E.Selcuk, “Implementatýon of a Distributed Shared Memory System”, Institute of Science and Technology, Ms.C. Thesis, 2000

[6] O.K.Sahingoz and N.Erdogan, "A Java Based DSM System for User Defined Shared Data Objects", Software and Hardware Engineering for the 21th Century, Editor N.E. Mastorakis, WSES Press, 1999

[7] K.Li and P.Hudak, "Memory Coherence in Shared Virtual Memory Systems", ACM Transactions on Computer Systems, Vol. 7,pp. 321-359, Nov. 1989.

[8] P. Keleher, A.L.Cox, and W.Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory", Proceedings of 19th Annual Int. Symp. On Computer Architecture,pp. 13-21, May 1992.

[9] K.Li,"IVY: A Shared Virtual Memory System for Parallel Computing", Proc.of 1988 Int. Conf. On Parallel Processing, IEEE Computer Society Press, Calif., pp. 94-101, 1988.

[10] P.Keleher et al., "TreadMarks:Distributed Shared Memory on Standart Workstations and Operating Systems" Proc.Usenix Winter Conf., Usenix Assos., Calif., 1994, pp.115-132.

[11] J.B.Carter, J.K.Bennet, and W.Zwaenepoel, "Implementation and Performance of Munin", Proc. 13th ACM Symp. Operating Systems Principles, ACM Press, New york,1991, pp.152-164.

[12] G. Cornell, C.S.Horstmann, Core JAVA, Sun Microsystems Press, 1997.