A Java Based DSM System for User Defined Shared Data Objects

OZGUR K. SAHINGOZ and NADIA ERDOGAN
İstanbul Technical University
Computer Engineering Department
Ayazaga, 80626, İstanbul, TURKEY

Abstract: *Distributed Shared Memory* (DSM) is a model for interprocess communication in distributed systems. A DSM system provides its users a simple, shared memory abstaction automatically, while message passing systems require data movement to be specified by the programmer. The focus of this paper is on software design and implementation of a Java based , user-level DSM system which facilitates sharing of user defined data objects across nodes in a distributed environment. Shared data is distributed across the system, using the *read-replication* (multiple reader/single writer) strategy. The *release consistency* model with *write-invalidate coherence policy* is adopted, where the owner of a shared data object is directly responsible for enforcing coherence. The system provides a set of primitives as a Java library, with which shared data objects are uniquely named and manipulated.

Key-words: distributed computing, shared abstractions, distributed shared memory algorithms, memory coherence, write-invalidate protocol.

1 Introduction

Shared memory is widely believed to provide an easier programming model than message passing for expressing parallel algorithms. Distributed shared memory (DSM) is a model for interprocess communication in distributed systems. In the DSM model, processes running on separate nodes can access a shared address space through normal load and store operations and other memory access instructions.(Fig.1.) The underlying DSM system provides its users with a shared, coherent memory address space. Each user process can access any memory location in the shared address space any time and see the value last written by any other process.

In recent years, the availability of high speed networks and high performance microprocessors is making networks of computers an appealing vehicle for cost effective parallel computing. Supporting the abstraction of shared memory on these high performance computers results in DSM systems which provide a shared address space over a message passing interconnect. The goal of the DSM system is to transparently run programs written for hardware shared memory systems. DSM systems have many advantages over message passing systems [1,2]. Mainly, they provide the user a simple, shared memory abstraction automatically, while message passing systems require data movement to be specified by the programmer. Many approaches have been proposed to implement DSM systems [3,4,5,6,7]. 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 [4].

The focus of this paper is on software implementation of a DSM system . It presents a Java [8] based DSM system which provides a set of primitives with which arbitrary shared abstractions can easily and efficiently be implemented across a distributed hardware platform. Our implementation uses the *read-replication* (multiple reader/single writer) strategy of distributing shared data across the system.The replication of shared data 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. Shared memory is structured as variable size user defined data

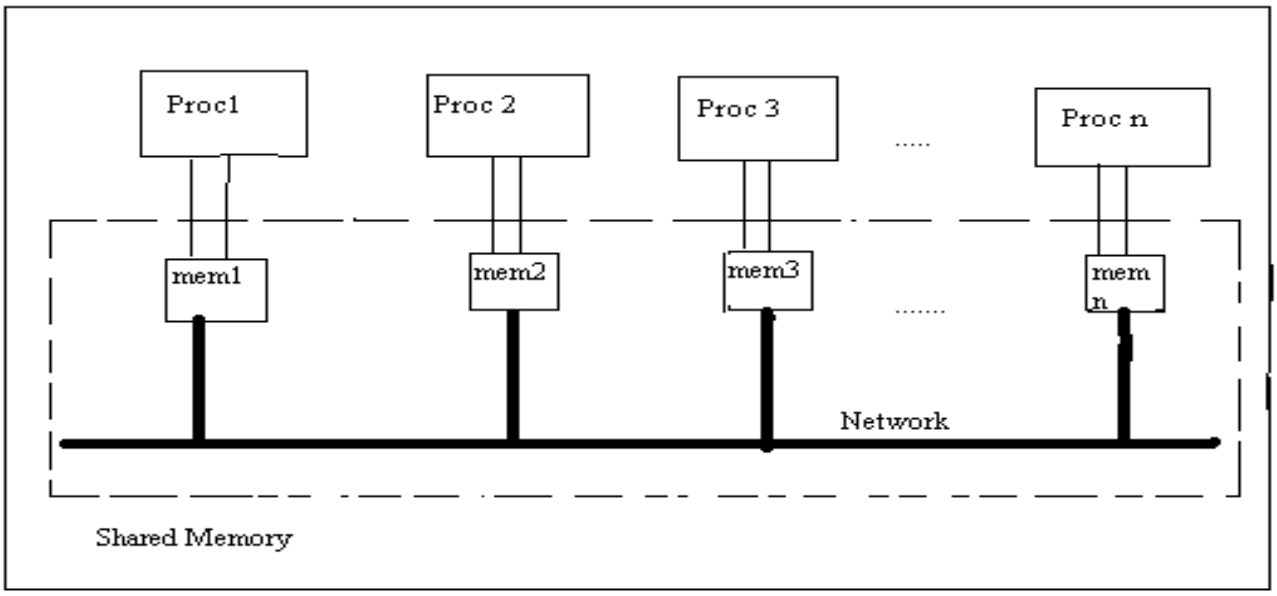

Fig.1. Distributed Shared Memory: Each process sees a shared address space, denotedby the dashed outline, rather than a collection of distributed address spaces.

objects in the source language Java.. Presently, only language defined types of variables are supported. Work is going on to include instances of user defined classes. The DSM mechanism is implemented at the user level, as a Java library for linking to application programs. It runs on an Ethernet network of Intel NT4.0 processors.

2 Information Structure

We assume an implementation where a shared data object directory is distributed among nodes and is organised as a hash table. Each node maintains an information structure for each shared data object which is either resident in its memory, or which is created on the local memory of another node but has been accessed by processes local to that node. An entry of the directory contains many pieces of information for each object, as its unique name, the local address, meta information (e.g., size, value) , etc. for each shared data item. The information in the directory is used to locate and transfer data objects, and to invalidate replicas.

Name: Two processes in an application share a data object if they call it by the same name. Therefore within each application, all shared data must be named uniquely across all of their replicated copies. The item's name is its unique identifier.

Owner: The identity of the unique owner node which owns the only writable copy of the shared data object and has the right to update that data 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 data object. This list is maintained by the owner node.

Probable Owner: Points towards the *owner* of a shared data object. When a node needs a copy of a data object, it sends a request message 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 a node having a copy of the data object is reached.

Status: A shared data block 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.

3 The DSM Algorithm: Read-Replication

The algorithms for implementing DSM deal with the problem of distributing shared data across the system.Two frequently used strategies are *migration* and *replication*. Migration implies that only a single copy of a data object exists at any time, so the data item 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 data 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)* [3] 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 a data 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 data item. 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 information kept in its DSM directory. For every read, write and invalidate request the probable owner field changes accordingly, to decrease the number of messages to locate the real owner.

The owner of data object also keeps its copy set, the list of nodes that have replicas of the shared data object. The copy set goes together with the data 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 The Coherence Policy: Release Consistency with Write-Invalidate Protocol

The replication of shared data blocks complicates issues in enforcing memory coherence. The coherence policy determines whether the existing copies of data item 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 data item 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 [7] is a *relaxed* memory consistency model that permits a node to delay making its changes to shared data visible to other processes until certain synchronisation accesses occur. That is, it allows views of shared data 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 data, but only one copy of a "writable" data object. All replicas of the shared data 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 data is available, it is returned immediately. If the data is not available, a read request is sent to the probable owner and a copy of the data is returned. The copy remains valid until an invalidation request is received.
For a write request: If the data block is writable, the request is satisfied immediately. Otherwise, a request for a copy of the data, 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 DSM Implementation

DSM management responsibility is distributed to all nodes on the system. The identity of the probable owner is kept for each data object. Requests are forwarded to probable owners until the real owner is located. The DSM algorithm is implemented by two processes, a *listener process* (LP) and a *message interpreter process* (MIP) present on each node of the DSM system (Fig. 2.).

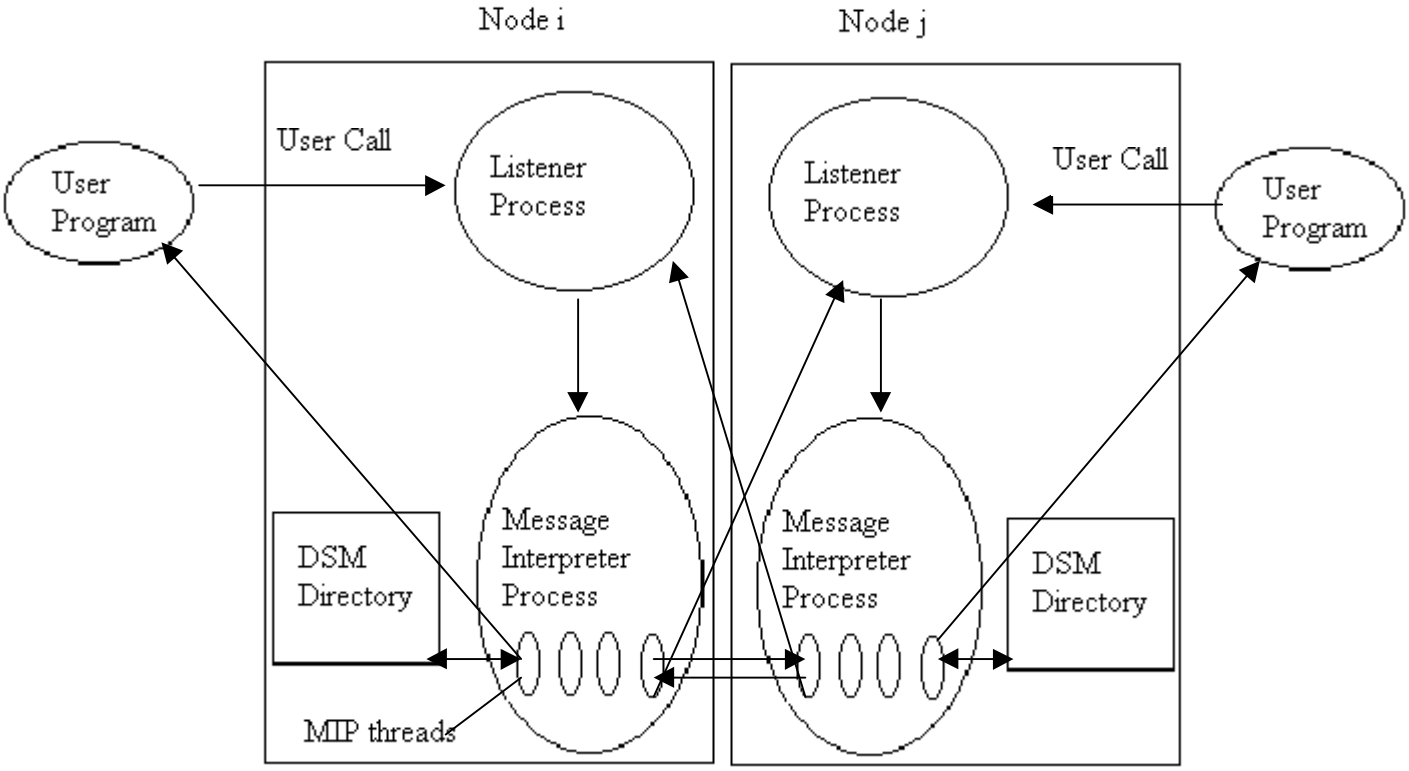

Fig. 2. DSM Implementation

5.1 Message Communication

Communication between DSM processes takes place through messages. Messages carry requests and resulting information between local and remote processes of the DSM system. 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 data about the shared object . Subfields of the message are evaluated differently, depending on the request type.

5.2 Listener Process

To handle message communications, this process listens for incoming messages on a specified port of the node. Sources of messages are either user processes executing locally on that node or remote threads of MIP processes. LP directly transfers the received message to MIP.

5.3 Message Interpreter Process

MIP is a *multithreaded process* which is responsible of DSM management. New nodes can be dynamically added to or removed from the DSM system. *Node list* is a data structure which keeps the identities of the nodes currently present on the system and is frequently used for *selective multicasting.* When a new node introduces itself to the system, MIP issues a broadcast message to all nodes on the network. Nodes which have already registered themselves to the DSM system acknowledge the new node with a reply message and also add its address to their node lists. The new node also builds up its node list with the addresses of the nodes from which it has received replies. Now, the integration of the new node to the DSM system is completed.

MIP creates a lightweight, *concurrent request handler* Java thread for each incoming request message. On receiving a request message from LP, MIP checks the type of the request and partitions the message contents accordingly, extracting the information necessary for its fulfillment. It creates a new thread for that particular request supplying the input data. Thus, multiple DSM requests are served concurrently, each by a different MIP thread. This improves program performance and modularity. All active threads have *exclusive access* to the local DSM directory to locate shared data. If the local directory does not possess the necessary information, then remote nodes have to be contacted. Threads use *selective multicasting* or *broadcasting* techniques depending on the type of the request, to contact

remote nodes. Selective multicast technique is used when nodes on the copy set of a data item are to be contacted. In this case, a thread, that sends messages to n nodes, has to wait for n reply messages. The thread determines an unused port and sends its identity together with the multicast message to the target nodes so that they can direct their reply messages to the that particular MIP thread which initiated the multicast communication. Establishing a direct communication between remote MIP threads improves performance by eliminating extra message traffic between DSM processes. A MIP thread terminates itself after it completes its task.

6 PRIMITIVES

The DSM system supports user defined shared data objects. Its basic functionality is comprised of a naming support and of calls with which shared data are manipulated. The DSM mechanism is imlemented at the user level, as a Java library for linking to the application program. The primitives which make up the user interface are discussed in detail below. Each call takes three parameters in the form :

Call(name, vartype, error)

where name is the unique name of the shared object, vartype is its type and error is a return value which reports the success or the cause of the failure of the request.

Create: The create call is used to initialize a shared data object. All shared data must be named uniquely. Naming as well as space allocation are achieved with this call. The user supplies a unique identifier and the type of the shared object. Presently, the basic data types defined in Java, namely, integer, string, char,boolean, float, long, byte, short and double can be used. We are currently working on having instances of user defined classes be treated as shared objects. On a create request, MIP refers to local and remote directories to check if the data object already exists. If not, an entry in the DSM directory is allocated for the data object and its ownership is assigned to the requesting process.

Remove: The remove call is used to purge a shared data object and remove it from DSM directories. Only the owner of a shared data object has the right to use this call. In case the requesting process does not possess the ownership of the shared object, MIP locates the object in the system, transfers its ownership and removes its directory entry, also invalidating other replicas.

Read: The read call is used to fetch and create a replica of a shared data object from a remote location to the caller's memory space. If the data is in available state in the local DSM directory, its value is returned. If the local copy is in unavailable state, a read request message is sent to the probable owner and a copy of the shared object is returned. In case no entry for the shared data exists in the local directory, MIP issues a selective multicast message to all nodes on the node list, to locate the owner of the data object and directs a read request message to that node to get a copy of the value of the object. The local copy remains in available state until an invalidation request is processed. Simultaneous local execution of read requests at multiple nodes is possible.

Read-w: The unique owner process of a shared object 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 a data object before it can issue a write request. The read-w call returns a valid copy of the data object together with its ownership so that the caller gets permission to update it. MIP sets the status of the object as *writable* and also sends selective multicast message to the nodes on the copyset of the data object. The message contains an *invalidation request* such that all replicas except the local copy are invalidated, passing into unavailable state.

Put: The owner process of a data object issues the put call to update its value. MIP tries to locate an entry for the object in the local DSM directory, as one should exist if the caller is the real owner of the data object. In case such an entry is not present, the call returns with an access error. Otherwise, the value of the shared object is updated and its state is changed to *available*, as it now possesses a valid value.

Lock: The lock call inhibits all accesses to the shared object, except the caller. The data object enters the *locked* state. It is used for synchronization purposes, to prevent data races.

Unlock: The data object changes into unlocked state so that read accesses may be again performed.

7 Conclusion and Future Work

This paper presents the design and implementation of a Java based, user-level DSM system that facilitates sharing of user defined data objects across nodes in a distributed system. The DSM system provides various primitives for data manipulation. The system is implemented on an Ethernet network of Intel NT processors. The performance of the system has been improved in several ways. False sharing results when a DSM system can not distinguish between accesses to logically distinct pieces of information. It can lead to situations where multiple processes contest ownership of a data block, even though they are modifying entirely disjoint sets of data. False sharing is common in systems which track accesses at the granularity level of virtual memory pages. Our system uses type and structure information for user defined data objects, at the granularity level of programming language variables. This prevents false sharing and also introduces significant gains in performance as small amounts of data are transferred. Another source of improved performance is the use of selective multicasting which reduces network traffic remarkably. The multithreaded implementation of the system also has a positive effect on the overall system performance. We are presently working on an extension to the system where instances of user defined object classes may be shared. We are considering using RMI (remote method invocation) for an efficient implementation.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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