# The DiSOM Distributed Shared Object Memory

# Position Paper

Paulo Guedes Miguel Castro Nuno Neves

INESC
R. Alves Redol 9, 1000 Lisboa, Portugal
email: {pjg,miguel,nuno}@inesc.pt
Tel: +351 1 3100-395
Fax: +351 1 525843

## 1 Introduction

DiSOM is a software-based distributed shared memory system for a multicomputer composed of heterogeneous nodes connected by a high-speed, low latency network [Guedes 93]. The current prototype comprises a Sun SPARCCenter 2000 with 10 processors and several SPARCStation 10, i486 PC and Dec Alpha, connected by ATM and Ethernet.

Programs in DiSOM are written using a shared-memory multiprocessor model where synchronization objects are explicitly associated with data items. Programs are composed of a set of parallel threads of execution. These threads share data objects and synchronize by explicit calls to system provided synchronization constructs. The system traps these calls and uses the information to drive both distributed synchronization and the memory coherence protocol. DiSOM uses the *entry consistency* memory model [Bershad 93] to ensure coherence. This model guarantees memory consistency, as long as an access to a data item is enclosed between an acquire and a release on the synchronization object associated with the data item.

DiSOM addresses two issues we believe to be crucial in distributed shared memory systems: good performance, achieved by a close integration of the programming model with the synchronization mechanisms, and fault-tolerance, with an efficient checkpointing algorithm that requires no extra messages during the failure-free period.

## 2 Integration of the Programming Model with the Synchronization Mechanisms

DiSOM provides a simple programming model consisting of multiple threads executing in a single shared address space. The basic entities in the system are arbitrarily sized language-level data items called objects, which exist in the shared address space. Parallelism is achieved with user-level threads which synchronize using locks, semaphores and barriers.

DiSOM uses the entry-consistency memory coherence model introduced by Midway [Bershad 93], that has been shown to reduce the number of messages required to ensure memory coherency in a number of applications. According to the entry-consistency model, data objects are associated with synchronization objects. Data objects are made consistent when a synchronization operation takes place on the associated synchronization object. In DiSOM the basic rules introduced by Midway were extended to correctly handle data objects that are associated with multiple synchronization objects and to support synchronization objects such as semaphores, that unlike locks, are not used with a pair of corresponding acquire–release operations from the same process.

A novel aspect in DiSOM is the way the synchronization primitives are integrated with the programming model.

Any synchronization object *sync* can have a list of data objects associated with it, and those objects are guaranteed to be made consistent when a synchronization primitive is invoked on *sync*. This eventually requires transmitting the objects between different nodes. To send and receive an object over the network, the system uses two call-backs defined by all objects, *pack* and *unpack*, to respectively marshal and unmarshal the object. A default version of these call-backs is automatically generated by our modified g++ compiler, and if necessary they can be easily customized by the programmer. This provides a very powerful mechanism to programmers, as they can specify *how* the object should be linearized to be transmitted, but they never have to explicitly specify *when* the communication should occur. That is left to the system, driven by the calls to the synchronization primitives.

This mechanism is used to transmit complex data structures (e.g. a sparse matrix) without increasing the number of messages. The call-backs *pack* and *unpack* of these classes know how to gather the data and construct a single message that is sent to the remote node, and there reconstruct the data structures from the message. A mechanism is also provided to describe parts of objects, making it possible to associate only part of a matrix to the synchronization object.

The same mechanism allows us to support heterogeneity. DiSOM supports heterogeneity at two levels – data item representations and address spaces. Different data item representations including different byte orders, different structure alignment and different memory layout of arrays are supported by DiSOM using an external data representation policy. DiSOM supports program execution using a conventional process in each node. The address spaces of these processes are not uniform in the sense that the objects are not mapped in the same addresses in all processes and the relative memory positions of the same objects vary from process to process. Nevertheless, the system guarantees that objects have unique names and are transparently shared, using the pointer swizzling technique. This solution allows each process to map the objects it accesses in arbitrary virtual addresses, using the most efficient representation generated by the compiler.

Some synchronization objects, such as barriers, are defined in such a way that they execute a call-back to user code whenever a meaningful synchronization message is to be sent or received. In the case of barriers, a call-back is performed on each participant when it arrives at the barrier and when it receives the message from the coordinator allowing it to continue; at the coordinator, a call-back is performed whenever a message is received from a participant, and just before sending the broadcast message to all participants. Each call-back can specify data to be piggybacked on the message to be sent, or to receive data that was piggybacked on the message just received. This very simple mechanism allows us to implement concepts such as voting, widely used in iterative algorithms to find out if all participants have converged, without any additional messages than those that would be necessary to implement the distributed synchronization. To the programmer they still look like conventional barrier objects, only with four additional call-backs.

# 3 Fault Tolerance

In this environment, the probability of a failure during an application's execution increases with the execution time and the number of workstations used. If no provision is made for handling failures, it is unlikely that long running applications will terminate successfully. In DiSOM, we have defined an efficient checkpoint protocol that allows transparent recovery from single node failures and, in some cases, from multiple node failures [Neves 94]. It is efficient because it is tightly integrated with the memory coherency protocol. It requires no extra messages during the failure-free period and it keeps a distributed log of shared data accesses in the volatile memory of the processes.

During the failure free period, processes keep in their volatile memories a log of shared data accesses and periodically but independently checkpoint themselves to stable memory. Each process logs the object versions it produces. A new version of an object is produced when a release-write operation is executed on a synchronization object to which the object is associated. When a process receives an acquire request for an object version it produced, it records in the log the execution point of the thread that made the request.

The first step to recover a failed process is to get its most recent checkpoint and reload it in a free processor. Next, the recovering process obtains all the object versions acquired by its threads between the checkpoint and the failure, from the logs in the other processes. Then, the threads of the recovering process start to re-execute their programs from the execution points at the moment of the checkpoint and to re-acquire the same versions of the same objects as they did before the failure. After recovery, in the event of a single process failure, the system will be in a consistent state. On the other hand, if multiple process failures occur, the checkpoint protocol may be unable to recover the system to a consistent state. The recovery procedure uses a simple mechanism to detect this situation and aborts the application.

# 4 Current Status

The current prototype supports applications written in C++ on a network comprising a Sun SPARCCenter 2000 with 10 processors and several SPARCStations 10 and Dec Alpha connected by ATM and Ethernet. Several applications have been written to test and evaluate the system, namely matrix multiplication, sparse matrix multiplication, successive over-relaxation and traveling salesman. Work is under way to port other applications from the Splash suite and real world applications for image rendering. The checkpoint protocol has been designed and is being implemented.

# References


[Bershad 93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 93 COMPCON Conference*, pages 528–537, February 1993.

[Guedes 93] Paulo Guedes and Miguel Castro. Distributed Shared Object Memory. In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, Napa, CA., October 14-15 1993. IEEE Computer Society Press.

[Neves 94] Nuno Neves, Miguel Castro, and Paulo Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. In *Thirteenth ACM Symposium on Principles of Distributed Computing (PODC)*, August 1994.