Experiences in Integrating Distributed Shared Memory with Virtual Memory Management*

R. Ananthanarayanan *Sathis Menon* *Ajay Mohindra*
Umakishore Ramachandran

College of Computing
Georgia Institute of Technology
Atlanta, Ga 30332
Ph: (404) 894-5136
e-mail: rama@cc.gatech.edu

Submitted to the 1992 International Conference on Distributed Computing Systems.
Also published as a technical report GIT-CC-90/40.

Abstract

While the duality between message-passing and shared memory for interprocess communication is well-known, the shared memory paradigm has drawn considerable attention in recent times even in distributed systems. Distributed Shared Memory (DSM) is the abstraction for supporting the notion of shared memory in a physically non-shared (distributed) architecture. It gives a uniform set of mechanisms for accessing local and remote memories. Further, by combining shared memory style synchronization with the actual data transport in the DSM design, it is possible to provide flexible sharing while assuring mutual exclusion when necessary. This research reports on our experience in implementing a DSM system in an object-based distributed operating system. The important observation from this experience is the need for a tight cooperation between the virtual memory management at each node and the mechanisms provided by the DSM. Such a cooperation is essential to assure good performance of the DSM mechanisms and make the shared memory paradigm viable in a distributed environment.

*This work has been funded in part by NSF grants CCR-8619886, and an NSF PYI Award MIP-9058430.

1 Introduction

Traditionally, shared memory and message-passing have been the two paradigms used for interprocess communication and synchronization in multi-process computations. The duality between the two paradigms for structuring computations is well-known [LN78]. Nevertheless, shared memory has been an appealing paradigm from the point of view of programming ease even in distributed systems as evidenced by the distributed programming environments that have been proposed based on this paradigm [LS83, JLHB88, Gel85]. We refer to abstraction of shared memory in a physically non-shared (distributed) architecture as *Distributed Shared Memory* (DSM). Several researchers [LH89, RAK89, BCZ90] have suggested system architectures to support this abstraction.

A strong motivation for DSM arises from the structure of current distributed computing environments, which typically consists of *compute servers* (workstations) and *data servers* (file servers). In such an environment, there are two chores to be performed to execute a computation. The first one involves selecting a compute server, which is a scheduling decision best left to a high-level policy making entity. The second one involves bringing the code and data from the data server to the selected compute server before executing the computation. This chore requires a remote paging facility. If sharing of data is coupled with this remote paging, it could be seen that DSM presents itself as a natural facility for combining the two.

When the shared memory paradigm is implemented on top of a physically distributed system, several issues need to be addressed. Firstly, a set of uniform mechanisms should be provided to access the shared memory regardless of whether it is local or remote. Clearly such mechanisms are needed to provide the transparency required at the user level. While such mechanisms may be implemented as library functions on top of an operating system, implementing system services is facilitated by providing these mechanisms at a fairly low level.

Secondly, there could be a potential mismatch between the communication abstraction provided by the operating system and the one required by the programming model, leading to poor performance [Sco87]. To support the shared memory paradigm, it is important that the abstractions chosen in the operating system match this paradigm. Further, care has to be taken in implementing these abstractions to reduce the overhead incurred at the low-level in terms of messages exchanged across the network.

Thirdly, virtual memory (VM) management is invariably an integral part of any general-purpose operating system [RTY+87, ARS89]. Therefore, remote memory accesses have to be reconciled with the virtual memory management.

The abstractions needed for efficiently implementing a shared memory paradigm in a distributed setting is the main thrust of this paper. Our experience shows that a tight cooperation between virtual memory management and distributed shared memory is essential to ensure good performance of this paradigm.

1.1 Object-Thread Paradigm

Object-based systems [ABLN85, DCM+90, Nor87] embody the shared memory paradigm: Conceptually objects are similar to an instance of an abstract data type. The interaction of independent computations inside objects usually involve shared memory style synchronization (such as locks and semaphores). *Clouds* is one such system and our experience in implementing the distributed shared memory is presented in the context of this system. However, the lessons we have learned from this experience have general applicability to the problem of implementing shared memory in a distributed environment.

Providing a unified environment over distributed hardware, location independence for data and computation, atomicity of distributed computation, and failure tolerance are some of the research goals of Clouds [DCM+90]. One of the distinctive features of this system is its separation of two notions that have been traditionally intertwined in most operating systems, namely, address space and computation. The former is specified by *objects* and the latter by *threads*. An object is a passive entity (i.e. there is no process associated with it) that is part of a global name space. Objects are persistent: once created they exist until explicitly deleted. It specifies a distinct protected address space that is unique in the entire distributed system. The object encapsulates code and data for executing *entry points* in the object that are *invocable* from other objects. To allow concurrent execution of more than one computation in the same object, shared-memory style synchronization primitives are provided by the operating system.

A thread[1] is an active entity that provides the notion of a computation. It executes in the context of an object. During the course of execution, a thread may invoke entry points in other objects. Thus a thread is not associated with a single address space. Further, since these objects may not all be at the same node, a thread may span machine boundaries during the course of its execution.

The collection of objects in this system represents a *distributed shared global virtual space*. A thread traverses the address spaces of the objects that it invokes during its execution. In this sense, the system represents a shared memory paradigm that is implemented on a collection of compute and data servers. In such a setting, invocation of objects on the same compute server depends on efficient virtual memory management; and invocation of an object that is not currently local to the compute server depends on the efficient manipulation of the distributed global virtual space. In addition, to support persistent objects DSM and VM have to unify

[1] Note that our notion of a thread may be different from other uses of the term.

secondary storage management with that of main memory management.

The rest of the paper is organized as follows. We present the system architecture for implementing the object-thread paradigm and the design rationale in Section 2. Salient features of the implementation are then discussed (Section 3), followed by performance results (Section 4). Reflections on our design to integrate virtual memory management and DSM is presented in Section 5, and concluding remarks are presented in Section 6.

2 System Architecture

In this section we present a system architecture for implementing the object-thread paradigm.

2.1 Local Virtual Memory Management

As noted earlier, invoking an object on the same compute server is virtual memory intensive. The design of the local VM subsystem has been influenced by three factors: First, each object has to be represented by a distinct protected address space. Second, a thread may have its own address space for holding thread-specific information such as a stack and a parameter passing area distinct from the object in which it is executing. Lastly, the kernel of the operating system needs its own protected address space. Therefore, the entire virtual address space provided by the machine architecture is partitioned into three distinct regions: O space for the object, P space for the thread, and K space for the kernel. These address regions are realized using a software abstraction called a *virtual space*. A virtual space is a monotonically increasing range of virtual addresses.

Storage in the system is represented by *segments*. A segment is an uninterpreted sequence (of arbitrary length) of bytes and is a container for a range of virtual addresses. Associated with a segment are static storage attributes (such as copy-on-write, persistent, non-persistent), and static access control information (such as read-only and read-write). A contiguous range of addresses in a virtual space is mapped to a segment. Each mapping is specified using a *window*. A virtual space is thus a collection of windows. A window also specifies the dynamic access characteristics of the segment that it maps.

A thread is a computational entity possibly spanning machine boundaries. Hence it needs a representation on the nodes that it visits. An *isiba* is a system abstraction that represents a computation at a node. A schedulable entity is realized by associating an isiba with either a K space or with all of O, P, and K spaces. The former is useful for realizing kernel daemons and the latter for realizing a thread. Object invocation involves un-installing the current O address region of the executing thread and installing the virtual space of the invoked object as the current O space. The P space of the thread is unchanged. On object return, the virtual space of the caller object is re-installed.

These abstractions – virtual space, segment, and isiba – are provided by the kernel. The conceptual layer above the kernel is a set of *system objects* that provide the essential functionality expected of an operating system such as page-fault management, buffer management, and device drivers. Segments are accessed by the kernel through a *partition* interface. A system object that provides this interface is called a partition. The partition interface specifies a minimal functionality that includes: *create*, *destroy*, *activate*, and *deactivate* of segments. Create function associates a system-wide unique name and a static type with a segment. It also associates the segment with an instance of a partition. This design allows several partitions to exist, each managing a collection of segments. Activate function sets up system data structures in preparation for accessing the segment. Destroy and deactivate provide the inverse functionality of create and activate, respectively.

The virtual memory subsystem uses the kernel abstractions and a set of system objects to provide an efficient machine independent virtual memory management. The set of system objects are: the virtual memory (VM) manager, the primary memory (PM) manager, the secondary storage (SS) manager, the swap manager, and the local partition. The calls provided by the VM subsystem are summarized in Table 1. These calls are used by the VM subsystem to service page-faults.

In realizing this VM subsystem on a real architecture, there is a need to map the machine independent structures to the machine specific memory management capabilities. We assume that a page is the fundamental unit of memory management in the underlying architecture. The (machine independent) segment is assumed to be composed of multiple pages.

Virtual Memory Manager

The most important function of the virtual memory manager is to make sure that a given page in a segment (referred to as *segment page* in the rest of this paper) is installed in physical memory. The VM manager translates the type of a page-fault to the appropriate access mode of the segment. Access violation checks are also performed by the VM manager. In addition, the VM manager recognizes the possible swapped status of a segment page and takes appropriate action through the swap manager.

Primary and secondary memory management

The physical memory at each compute server is divided into frames. These frames are managed by the primary memory manager. The PM manager provides for allocation and deallocation of the physical pages. It is also responsible for realizing a page replacement policy. The swap manager provides swap-in and swap-out operations for use by the VM and PM managers. It maintains the correspondence between the pages belonging to the segments and the swapped out physical frames. The secondary storage manager provides backing store for segments.

Local Partition

The local partition is a system object that manages non-shared segments that may fall in one of the two categories — *local* or *perishable*. Local segments are those which are not shared between compute servers but need to be persistent. For example, these segments may be used to save the state of a system object across node crashes. Perishable segments are used to represent memory which is non-persistent and non-shared. For example, the thread stack and the parameter passing area may use perishable segments.

2.2 Distributed Shared Memory Subsystem

The VM subsystem discussed thus far provides the mechanisms for invocation of objects local to the compute server. The abstraction that provides the mechanism for performing invocation on objects that are remote to the compute server is DSM. While a local partition provides support for local and perishable segments, the DSM subsystem provides support for shared segments. The DSM subsystem has two parts: the DSM controller (DSMC) and the DSM partition (DSMP). The former is a "server" system object for remote requests and the latter is a "client" system object that the VM subsystem contacts to access a remote segment.

DSM Controller

There is a concept of *ownership* of a segment and the DSMC where the segment is created is responsible for guaranteeing the consistency of the segment. The basic operations provided by the DSMC are `get` and `discard`. The `get` operation is used to fetch a segment from its owner, while `discard` is used to return a segment to its owner. The DSMC provides combined access and lock operations using the `get` and `discard` primitives. Detailed algorithms for maintaining the consistency of DSM are presented in an earlier paper [RAK89]. Using the `get` primitive a segment may be acquired in one of four modes: `read-only`, `read-write`, `weak-read`, or `none`.

`Read-only` signifies non-exclusive access but guarantees that the segment will not change until the thread explicitly discards it.

`Read-write` signifies exclusive access (for the thread) with a guarantee that the segment will not be given to any other thread until the thread explicitly discards it.

`Weak-read` signifies non-exclusive access with no guarantee whether the segment will change or not. The owner DSMC immediately honors a `weak-read` request by sending a copy of the segment to the requester. In this mode, the requester may choose to receive future updates to the segment.

`None` signifies exclusive access with no guarantee whether the segment will be yanked away or not. If the segment is currently held in `read-only` or `read-write` modes then `none` requests are enqueued in the appropriate segment queue. If the segment is available at the owner DSMC, it sends the segment to the requester. The requester becomes the *keeper* of the segment and the owner remembers the current keeper. If the segment is cached at another node in mode `none`, then the owner DSMC instructs the current keeper to forward the segment to the requester. A segment cached in mode `none` at a keeper node may be returned to its owner by issuing a `discard`.

In addition to the data transfer operations mentioned above, the DSMC provides the following control operations on segments that it owns corresponding to the generic partition interface: create, destroy, activate and deactivate. All of these operations are performed on a per segment basis.

Read-only and read-write modes assure strong coherence for shared segments by implementing a multiple-readers/single-writer policy. However, such strict coherence may not always be required. Weak-read mode allows tailoring the coherence requirements in an application specific way. None mode ensures that there is exactly one copy of a segment at all times in the system, and therefore has limited coherence maintenance overhead.

DSM Partition

The DSMP conforms to the generic partition interface and is used by the VM manager to service a page-fault to a shared segment. The DSMP provides methods mirroring the data transfer and control operations of the DSMC. In addition, the DSMP may also perform various optimizations that are designed to reduce the data transfer overhead by taking advantage of the segment access modes supported by the DSMC (see §3.2.1). The DSMP interacts with remote DSMCs through a communication subsystem (see §2.3). When a `get` primitive is to be issued, the DSMP sends a request to the owner DSMC and suspends the requesting process until the segment is received.

2.3 Communication Subsystem

The DSMC, the DSMP, and the SS manager at the compute servers, and the DSMC and the disk server at the data servers need to communicate with one another reliably across the network (see Figure 1). The communication requirement is a simple reliable request/response type of protocol. In our implementation, we use a transaction-oriented message-passing protocol (RaTP) [Wil89], similar to other transaction-oriented protocols such as VMTP [Che86]. However, it is much simpler than VMTP since it is tailored to our requirements. The transactions consist of *request* (a blocking transaction), *reply* (a non-blocking transaction), and *standby* (a blocking transaction used for polling requests or replies).

2.4 Interaction Between DSM and VM Subsystems

Objects and segments constitute the two dimensions of an address space that spans the entire network. When an object is accessed by a compute server that is not currently mapped in its physical memory, a *page-fault* occurs. To service this fault, the page has to be brought into memory by querying this 2-dimensional address space.

Figure 1 shows our distributed memory management architecture. Given this architecture, there are two ways, *explicit* and *implicit*, to make a segment accessible to an executing thread. The explicit way is to prefetch the segment before it is needed. This method may be used for (a) reducing the page-faults by prefetching some (page of) segments before scheduling a thread, or (b) acquiring the segments in a particular lock mode allowed by the DSMC. The implicit way is to wait until a page-fault occurs. Since the interaction between the components of this architecture is similar under both scenarios, we consider only the implicit case in this discussion.

When a page-fault occurs, the hardware address space corresponding to the faulting address is discerned. The address space is one of O, P or K spaces and is backed by a corresponding virtual space. From this virtual space, the window structure that maps the faulting address is determined. From the window, the name of the segment and the page offset within the segment is computed.

A page-fault could occur in one of the following three cases. The first type of page-fault occurs on a segment page which has never been accessed (paged-in) before. This type of page-fault has to be serviced by contacting the appropriate partition. For example, if the segment belongs to a DSMP, then the VM manager contacts the DSMP. The DSMP in turn, sends a request for the segment to the appropriate remote DSMC. When a reply is received from the DSMC, the DSMP transfers the received segment page to the page frame (preallocated by the VM manager using the PM manager) and returns control to the VM manager. The VM manager passes control to the kernel which sets up the hardware MMU maps and restarts the faulting process.

The second type of page-fault may arise because of remapping a previously mapped segment to a new virtual address. This implies *sharing* of the segment page. In this case, the VM manager notifies the DSMP to ensure that such sharing is consistent with the lock mode that is currently associated with the cached segment. The third type of page-fault occurs on a segment page which has been swapped out. To service this fault, the virtual memory manager checks to find out if the required block has been previously swapped out. If so, the swap manager is notified and the page is swapped back into a new page frame obtained through the PM manager.

2.5 Design Rationale

2.5.1 Virtual Memory

The choice of the fundamental virtual memory abstractions, namely segment and virtual space, was directly influenced by attributes associated with objects (such as protection and persistence). Similarly, the backing store is a general facility that is needed for servicing page-faults (to fresh or swapped out pages) in addition to serving as a repository for persistent data. Therefore, we decided that the notion of a backing store should be divorced from any specific operating system functionality such as page-fault handling. Further, to ensure modularity of design we decided to encode the supporting functionalities (such as partitions) to the virtual memory abstractions in system objects rather than making them part of the kernel. At the same time we did not want to pay a performance penalty due to having partitions outside the kernel. Therefore, system objects have been implemented in the kernel address space.

A partition is responsible for managing segments. In the case of persistent segments a unique partition is assigned the responsibility for determining the backing store that has to be contacted to page in the segment. Similarly, non-persistent segments are realized by another unique partition. This design allows having several partitions for managing segments. There are two advantages to this design: First is the increased potential for parallelism in servicing simultaneous requests. The second is the possibility for organizing segment management by types (for example, shared segments are managed by the DSM partition). A local partition avoids network and DSM coherence overhead and may be used for perishable segments.

The VM abstractions provide uniform access mechanisms for any combination of local, remote, persistent, or non-persistent memory. A single virtual space may be composed of several windows that map to these different kinds of memory (through segments). As noted earlier in §2.4, when a page-fault occurs, the fault is serviced uniformly by the VM subsystem. Higher level mechanisms such as object invocations are oblivious to the type and location of memory that is involved in servicing the fault.

Mach [RTY+87] and Chorus [ARS89] are two systems that have demonstrated the importance of machine independent virtual memory management. Our VM design [BHK87] which evolved around the same time as Mach embodies similar principles. The fundamental difference between these other systems and ours is the design goals. Portability to new architectures and efficient implementation of Unix on their kernels are among the design goals of both Mach and Chorus. The way shared pages are handled on Unix "fork" is an attestation to the amount of thought that went into both systems to provide efficient support for Unix. The main difference between Mach and Chorus insofar as the virtual memory management is the way they handle versions of shared pages when they are written to by different threads. Supporting user-defined memory management is another important goal of Mach.

On the other hand, our virtual memory abstractions have been designed for efficiently supporting the persistent passive-object and active-thread paradigm. Thus, for example, a write to a shared page does not necessarily involve shadowing as it may in Mach and Chorus since the semantics of object-thread interaction does not necessitate such shadowing. However, we do support copy-on-write semantics at the segment level to allow efficient implementation of orthogonal system services such as support for recovery and fault tolerance.

Our segment abstraction is quite similar to the memory object of Mach. The difference is that a memory object is user-definable while a segment is not. Further, a segment represents a single level store and has persistence as an attribute. In Mach, the semantics of the memory objects are defined entirely by the user. The functionality of the partition system object in our design is similar to the external pager of Mach. The important difference is that while the external pager is in the user space, the partitions are in system space. We made this decision for efficiency considerations since every object invocation involves mapping and unmapping of segments which results in consulting the partition.

2.5.2 DSM Design Issues

In designing a DSM system, an important point to note is that the nodes (compute and data servers) do not physically share memory. The local memory at each node may be considered simply a cache of the global address space that spans the entire network. The DSM and VM management at each node would have to cooperate to ensure that the semantics implemented by the DSM manager and the VM manager are not compromised. The normal VM chores such as page replacement, swapping, and flushing have to be done in consideration with the DSM algorithms. Similarly, in satisfying a remote memory request the DSM would have to consult the VM manager to get a page frame, etc. Upon release of a page the DSM has to instruct the VM manager to invalidate page table entries, and take other related actions.

The effectiveness of the DSM paradigm depends crucially on how quickly a remote memory access request is serviced, and the computation is allowed to continue, which in turn depends on several factors:

• the delay before the VM system detects that a memory access fault (i.e. a page-fault), or a prefetching request entails a remote access

• the software overhead involved in the DSM protocol (i.e. coherence mechanism) for servicing a remote memory access request

• the software overhead involved in the communication subsystem (i.e. the basic transport protocol) for effecting the inter-node message communication to service the request

• the speed of the communication medium (i.e. hardware).

In addition to virtual memory management, there are several other issues that need to be considered in designing a DSM system.

Data Granularity

DSM systems add a new dimension to the memory hierarchy, namely, remote memory access across the network. The choice of the network plays a big role in determining the latency. Nevertheless, independent of this choice there is a fixed software overhead to be incurred depending on the choice of the data transfer protocol on the network. Moreover, such remote memory accesses need to be integrated, as noted above, with the memory management at each node. The requirement to integrate VM management and DSM often forces the granularity of access to be an integral number of the fundamental unit of memory management (usually a page). However, it is possible to reduce the network latency by transferring the page partially.

Memory Model

Lamport [Lam79] has proposed *sequential consistency* as an ordering constraint for the correct execution of a multiprocess program on a shared memory multiprocessor. Essentially, sequential consistency ensures that the view of the memory is consistent at all times from all the processors. However, it has been felt that sequential consistency is overly restrictive [DSB86]. It is possible to weaken the coherence requirements considerably and even tailor it to an application. Such weakening of the memory model is an attractive approach to making the DSM design more viable.

Choice of Protocol

To maintain the consistency of DSM, one of three approaches may be adopted :

1. **Write-broadcast policy**: This is the most intuitive scheme wherein a write to a location is sent to all the nodes that have a copy, thus updating all the copies. While this scheme is simple to implement, it is expected to generate considerable network traffic.

2. **Write-invalidate policy**: In this scheme instead of updating, the first write to a location causes invalidation of all the other copies. The traffic on the network would depend on the level of sharing with this scheme. A major disadvantage of this scheme is the unnecessary invalidations that would have to be sent to nodes that are no longer interested in a particular location even though they may have it cached.

3. **Lock-based policy**: In this approach, a request to read or write an item is preceded by a lock-request for the same item, and succeeded by a lock-release. By combining the lock-request (synchronization) and the request for the item, it is possible to provide synchronization for free with this approach.

In distributed systems, the number of messages is a measure of protocol performance. From this standpoint, the lock-based policy is expected to out-perform the other two, since coherence is maintained commensurate with the semantics of sharing in the computation. Moreover, since locking could be integrated with the data transfer there is no need for any additional mechanisms for providing mutual exclusion for shared write accesses. In both write-broadcast and write-invalidate policies there is a need to provide synchronization mechanisms on top of the coherence policy to assure mutual exclusion for multiple nodes requesting to write to the same page. However, lock-based policy has its drawbacks: In particular it does not have the generality of the other two policies. By decoupling memory coherence and synchronization, it is possible to devise synchronization mechanisms independent of the coherence policy. The lock-based policy requires explicit directives from the system software to know the semantics of sharing, while the other two do not require any such directives.

Synchronization

Finally, DSM systems need to provide some form of synchronization primitives to control access to shared data. These primitives may be built on top of the DSM or integrated with coherence maintenance.

In our design we provide strict memory coherence through the `read-only` and `read-write` modes. We also allow the user to tailor the coherence requirements through the `weak-read mode`. The unit of coherence maintenance is a segment and synchronization is built into the coherence maintenance. For efficiently managing remote paging and sharing, we decided that a tight integration of the VM and the DSM is necessary. A qualitative comparison of existing DSM systems can be found in [NL91] and [MR91].

3 Implementation Details

The salient features of implementing the VM and DSM subsystem are presented in this section.

3.1 Local Virtual Memory Structures

Segments

In their active state, segments are represented in the system by an *Active Segment Descriptor* (ASD). An ASD (see Figure 2) is a data structure containing the following information:

- Protection Attribute: static protection associated with a segment at the time of creation.
- Type: *ZeroFill*, *CopyOnWrite*, *Persistent* or *Perishable*; interpreted by the partition.
- Partition: points to the partition controlling the segment.
- Map: keeps track of the physical frames corresponding to the pages of the segment.
- Window List: list of windows (possibly in different virtual spaces) that map this segment.

Though segments are expected to be usually small, the ASD should have the ability to support large-sized segments, and allow for efficient look-up of physical frames corresponding to a segment page. Therefore, we chose a multi-level directory structure similar to the *inode* structure of the UNIX file system. An ASD is initialized on activation of a segment through its controlling partition. When a segment page has to be brought into memory, the segment map is updated to reflect the page frame corresponding to the segment page. Later, this information may be used by the partition to write the pages of the segment back to secondary storage. This write-back is made possible by the kernel *flush* operation on a segment, which writes out all the pages belonging to the segment cached in the physical memory. The segment map is also used to store the swapped status of the segment pages.

Virtual Spaces

A virtual space is represented by a data structure called the *virtual space descriptor* (VSD), which consists of a set of windows. Each window specifies the starting virtual address, the segment name the virtual address range maps to, the offset from the beginning of the segment where the mapping starts, and the length of the virtual address range that is mapped by this window. A VSD is stored as a segment and it has to be mapped into memory before any operations could be performed on it. The kernel provides special operations to map VSDs into reserved areas in the kernel. Operations supported on the VSD include determining the window mapping a given virtual address and adding/deleting mappings for address ranges.

Primary Memory Manager

The PM manager maintains the status of every physical frame in the system by using a software data structure called the *pframe*. The pframe data structure maintains the mapping between the machine independent abstraction (segment) and the machine dependent structures (MMU contexts and page table entries). The pframe also keeps a reference count to enable garbage collection. As a special case, a pframe may be *raw mapped* (not part of any segment) to an address in the K space. Raw mapping is useful for efficiently sharing data between system objects (§3.2.1). The PM manager in association with the kernel provides calls for mapping and unmapping a pframe structure.

Secondary Storage and Swap Managers

The SS manager provides a stable storage abstraction at each compute server. In our current implementation the compute servers do not have any local disk. We use the UNIX filesystem as a backend to provide this service through disk servers that run on top of UNIX and communicate with SS manager on the compute server through RaTP, as shown in Figure 1. The swap manager is used by the PM manager and the VM manager. When the PM manager runs out of free physical frames, it selects a victim, contacts the swap manager to swap it out, and invalidates the physical frame. Using the information in the pframe, appropriate kernel routines are called to indicate that the page has been swapped out. The VM manager may use this information in the future to service a page-fault on a swapped out page.

3.2 DSM Structures

The DSMP maintains information about segment pages that are cached-in at a node in a data structure called the `Dtable`. Each segment is represented by a `DtableEntry` (DTE). Each DTE contains the following information:

- segment name
- pages of the segment cached
- mode in which the segment was acquired
- a field indicating if an earlier request on this page is in progress

This information could have been maintained in the ASD for the segment itself. But the mode information is specific to DSM, whereas the ASD maintains general information about a segment. Hence we use a parallel structure. DTE's are allocated whenever a page is brought into memory, through the DSMC `get` primitive, and the appropriate fields of the DTE are initialized.

On discarding the page, the DTE is unallocated. On a page-fault, the VM subsystem allocates the pframe before requesting the DSMP to page in the segment; whereas for explicit segment requests, the DSMP acquires a pframe from the PM manager, and instructs the VM manager to install the pframe into the appropriate virtual space upon the arrival of data into the pframe.

The DSMC maintains state information about the segments it manages in a table called the `Ctable`. Similar to the `Dtable`, each segment is represented by a `Ctable entry` (CTE), which contains the following information:

- segment name
- mode in which the segment page has been handed out
- pages of the segment that have been handed out
- number of readers holding the segment in mode `read-only`
- list of requesters waiting for this segment
- node (keeper) with the copy of the page in mode `none`

The DSMC at each node is responsible for handling requests to the segments it owns as well as the segments that are currently cached at this node. There could be "forwarding" requests from the owner DSMC for segments that are currently cached at a node. For such requests, the DSMC in cooperation with the VM subsystem invalidates the pages of the segment that are cached, deletes the DTE corresponding to this segment, and sends the requested pages directly to the requester. For further details, the reader is referred to [RK89].

3.2.1 Optimizations for DSM

Several optimizations have been implemented to reduce the overhead of coherence maintenance and remote paging. The first one concerns subsequent `read-only` requests to a segment that is currently cached at a node in the same mode. The new request could be serviced without data transfer, if there is no pending request in `read-write` mode at the owner DSMC. While making the new request, the DSMP lets the DSMC know that it has the segment in `read-only` mode. If there is no pending `read-write` requests, the DSMC grants permission to the DSMP without any data transfer.

A second optimization is to reuse data associated with discarded segments. By maintaining version numbers with segments at the DSMP and DSMC, a segment request to the DSMC encloses this version number. If the version numbers match then the DSMC may grant permission to the DSMP without any transfer of data.

A third optimization is to recognize a subsequent request for a page while a previous one is still pending for the same page. These requests may be blocked until the original request completes. Once the original request completes, it may be possible to apply the first optimization to the blocked requests.

3.3 Page Fault Service — An Example

This section presents an example to illustrate how the data structures in the implementation are manipulated to service a page-fault. Let the user's view of the virtual address space range from 2000 to 6000 as shown in Figure 2. The representation of this virtual space is a VSD specifying the mapping. To service a page-fault at some location (4000 in this example), the kernel examines the corresponding VSD, and determines the window that maps the faulting address. As shown, this window maps virtual addresses starting at 4000 to a segment named `A`, starting at zero offset to a length of 2000. Thus, the pair {`A`, 0} is the effective page address at this point. From the segment map in the ASD for segment `A`, it is determined that there is no pframe mapping the required page. Before the controlling partition is contacted to fetch the page, the VM manager confirms that the access is valid by comparing the page-fault type and the protection attribute in the window. Let us assume that the fault type is due to a *write* and that the VM manager implements single-copy semantics for shared writable segments. This policy would translate to mode `none` for the DSMP. The VM manager allocates a pframe, and the DSMP is issued a `get` primitive with the triple {`A`, 0, `none`} to fetch the required page. The DSMP sends a request to the appropriate DSMC and obtains the page. A DTE is allocated in the `Dtable` to store information for the page in segment. Then, the segment map in the ASD for segment `A` is updated with the pframe storing the page. To complete the page-fault service, MMU mappings are set appropriately to point to the pframe, using the machine dependent kernel routines (see Figure 3).

4 Performance Results

The performance measurements reported in this section are done on a configuration of Sun 3/60s connected by a 10Mbps ETHERNET. A Sun 3/60 has a computing capacity of 3 MIPS (MC68020, 16 MHz clock). The timings are taken on a compute server (Sun 3/60) using a microsecond timer [DM90]. The reported numbers are the average of 10,000 measurements. A page in this section refers to 8K bytes. We report the performance results for three categories of experiments.

The first category of experiments is basic system timings, summarized in Tables 3 and 4. The tables also show a breakdown of the measured total time in terms of the low-level chores that constitute a system function. All the processing times in the table are measured numbers (by instrumenting the kernel) while network latency (wire overhead) is computed from the amount of data transmitted and the network bandwidth. The context switch time of 0.15 milliseconds involves switching between two different isibas. This switching entails saving registers, and other state information associated with an isiba and installing the new isiba on the processor. There is very little MMU overhead associated with this switching since all the kernel isibas exist in the same machine address space. All the network communication times shown in Tables 3 and 4 are between two compute servers. A null round-trip message (64 bytes) between two isibas making use of the ETHERNET system object takes 1.59 milliseconds. The RaTP level null round-trip time is 3.56 milliseconds. Given this null round-trip time, a page transfer takes 12.3 milliseconds at the RaTP level as it breaks up an 8K byte message into 6 packets (Table 4). Note that ETHERNET allows a maximum packet size of 1532 bytes [SDRC82].

The second category of experiments exercises the DSM subsystem, that builds on the basic timings. Table 5 summarizes the results and gives a breakdown of costs for the `none` mode requests. Other modes would incur the same cost if there were no queueing delays at the servers. A `get` from a data server takes 15.5 milliseconds. Comparing the DSM and RaTP timings (Tables 4 and 5) for a page transfer, it can be seen that the DSM protocol has an overhead of 3.2 milliseconds. This overhead includes updating state information for the shared segment and coherence maintenance. A `get` with forwarding incurs an additional overhead of 3 milliseconds over the simple DSM transfer due to an extra message being exchanged between the owner and the keeper of the segment (§3.2).

The third category of experiments (Table 6) deals with the servicing of page-faults. In the case of remote page-faults, there is no disk access involved (i.e. page is in memory at the remote server).

- A segment is currently with the data server that owns it. A DSMP on a compute server requests a page from that segment while servicing a page-fault. The average time for servicing such a page-fault is 16.3 milliseconds.

- A segment is currently in use at a compute server. To service a page-fault for this segment at another compute server, one level of forwarding (from the data server that owns the segment) of the request is involved. This three way exchange of messages results in a page-fault servicing time of 19.3 milliseconds.

It should be noted that the VM overhead of installing a page once a DSM `get` completes is only 0.800 milliseconds (difference between Tables 5 and 6). The last two entries in Table 6 show the time for servicing a page-fault on a segment owned by the local partition. Such faults do not require network messages, resulting in a time of 1.52 milliseconds for a zero-filled page and a time of 0.65 milliseconds for a non zero-filled page.

4.1 Comparison to Other Systems

As Schroeder and Burrows [SB90] point out it is extremely difficult to compare measured performance of systems due to the differences in the communication primitives (such as RPC, or unstructured messages, or remote memory requests), the level (such as kernel or user) at which the measurements are taken, the details of the implementation (such as emulation on top of some other operating system, or implementation on native hardware), and the choice of processors, the network and any other special purpose hardware. Hopefully the breakdown we have given in the previous subsection would convince the reader that we are not spending an undue amount of time in any activity related to the implementation. However, to make this work complete we present the reported performance of a few other systems just to put our measurements in perspective. The objective here is merely to show that our measured performance is in the right ball-park.

In Table 7 we give the applicable numbers from other systems with respect to ours. Time to create a memory region, zero-filling it and destroying it takes 3.12 milliseconds in the Mach system while 1.50 milliseconds in the Chorus system. An equivalent of this in our design is to activate a segment, service the page-fault with a zero-filled page and deactivating the segment. The cumulative time to perform these operations is 2.15 milliseconds (including a 1.52 milliseconds for servicing a zero-fill page-fault) in our system. All three timings are on comparable hardware. Using the IPC mechanisms of Locus operating system, fetching a remote 1K byte segment across the network takes 27.5 milliseconds in the Mirage [FP89] system. Mirage has been implemented on top of of VAX 750s connected by 10 Mbps ETHERNET. A similar fetch of an 8K byte page in our DSM implementation is 15.5 milliseconds.

Servicing a remote 8K byte page-fault takes only 13.1 milliseconds in the Amoeba system while it takes 16.3 milliseconds in our system. Both systems use comparable hardware. While the disparity is not much the difference may be attributed to two reasons: First, the basic transport protocol of Amoeba is better in performance; second, the Amoeba system does not incur the coherence maintenance overhead that is inherent in our system.

5 Reflections on the Integration

We made a conscious decision in the design of our system architecture to integrate the virtual memory mechanisms and the DSM mechanisms. In retrospect we feel this decision has paid off in several ways:

- From the point of view of virtual memory management, a remote page-fault may now be treated simply as an implicit remote memory request serviced by the DSM. Given that remote paging is a reality with the structure of current distributed environments (see Section 1), our decision to integrate the two paid off in this regard. The effectiveness of handling remote page-faults in this manner relies on how quickly the VM subsystem recognizes a page-fault as an implicit request for remote memory to be handled by DSM. The typed segments and the partitions allow this recognition to happen swiftly. We do pay a small price in that coherence maintenance is incurred for remote page-fault service which could have been avoided if page-faults were handled differently. But the price is not much as has been pointed out in Section 4.

- The VM manager translates remote page-faults to the appropriate modes of the DSM `get` requests, thus providing implicit locking when needed. One disadvantage of the integration is that the VM subsystem is reliant on the mechanisms provided by the DSM. However, experimenting with new protocols is not impeded by this decision so long as the DSM retains the same external interface.

- Since remote memory requests (through the DSM) have to be reconciled with virtual memory management it is imperative that we integrate the two from the point of view of performance. This integration allows the DSM to pre-allocate memory buffers while initiating the remote request, and avoids any unnecessary copy overhead to/from the communication buffers. Once the request is complete, the DSM directly instructs the VM manager to install the segment in the appropriate virtual space.

- The DSM allows sharing of segments (in `read-only` mode) between independent threads. The ability of the VM subsystem to map the same segment in multiple virtual spaces makes this sharing possible.

- The optimizations in the implementation of DSM (see Section 3.2.1) to reduce the message transfer overhead would not have been possible but for the tight cooperation between the VM subsystem and DSM.

- Certain functions such as object mobility becomes naturally available as a result of providing the DSM functionality at the lowest level and making it available to the VM subsystem [RK91].

- Similarly, migrating a computation to a different compute server (for reasons of load balancing) is made extremely simple. Due to the system-wide unique name for a segment, the appropriate segments automatically appear (due to page-faults) at the new compute server where the thread is re-started. The information that needs to be shipped at the time of the thread migration is an isiba context and the virtual space descriptors for the thread and the object. It can be seen qualitatively that very little overhead would be incurred in the thread migration [Zay87]. It is worth noting that without making any conscious design decision to support thread migration, we achieve it in our system.

6 Concluding Remarks

Distributed shared memory is a popular abstraction since it is an appropriate vehicle for adapting the shared memory paradigm to a distributed system. However implementation of this abstraction requires a careful evaluation of several design issues such as interaction with virtual memory, data granularity, memory model, choice of coherence protocol, and synchronization. In particular, since every node in the distributed system is an autonomous entity in terms of managing its local resources (such as processor and local memory) it is important to reconcile the implementation requirements of DSM with the mechanisms local to each node. In this paper we presented our experience in implementing a DSM subsystem in an object-based distributed operating system which represents a shared memory paradigm through the interaction of objects and threads. An integrated system architecture that uniformly handles requests for different types (such as persistent and non-persistent) of memory, independent of its location (local or remote to the node) was presented. The salient features of the implementation were discussed and measured performance for local and remote memory operations were presented. Our experience shows that an efficient implementation of DSM is possible with a tight integration of the DSM mechanisms and the virtual memory mechanisms.

Once such an integration is achieved, the two significant sources of overhead that remain in a DSM system are the coherence protocol processing overhead and the latency on the wire. Our directions for future research include providing hardware support for alleviating the processing overhead as well as using fiber optic interconnects for reducing the network latency.

References

[ABLN85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, January 1985.

[ARS89] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic virtual memory management for operating system kernels. *Symposium on Operating System Principles*, pages 123–135, 1989.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *The 17th Annual International Symposium on Computer Architecture*, May 1990.

[BHK87] José M. Bernabéu Aubán, Phillip W. Hutto, and M. Yousef A. Khalidi. The architecture of the Ra kernel. Technical Report GIT-ICS-87/35, School of Information and Computer Science, Georgia Institute of Technology, December 1987.

[Che86] D. R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of SIGCOMM '86*, August 1986.

[DCM+90] P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. LeBlanc, W. F. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The Design and Implementation of the Clouds Distributed Operating System. *The Journal of the USENIX Association*, 3(1):11 – 46, Winter 1990.

[DM90] P. B. Danzig and S. Melvin. High resolution timing with low resolution clocks and a microsecond resolution timer for sun workstations. *Operating System Review*, 24(1):23–26, Jan 1990.

[DSB86] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *The 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[FP89] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. *Operating Systems Review*, 23(5):211–223, Dec 1989.

[Gel85] D. Gelernter. Generative communications in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb 1988.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transaction on Computers*, C-28(9):690–691, Sep 1979.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321 – 359, Nov 1989.

[LN78] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. In *Proceedings of the Second International Symposium on Operating Systems*, Oct 1978. Reprinted in Operating Systems Review, (13,2) April 1979.

[LS83] Barbara H. Liskov and R. W. Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, pages 381–404, Jul 1983.

[MR91] Ajay Mohindra and Umakishore Ramachandran. A survey of distributed shared memory in loosely-coupled systems. Technical Report GIT-CC-91/01, Georgia Institute of Technology, Atlanta, GA, Jan 1991.

[NL91] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, pages 52–60, August 1991.

[Nor87] J. Duane Northcutt. *Mechanisms for Reliable Distributed Real-Time Operating Systems*, volume 16 of *Perspectives in Computing.* Academic Press, 1987. Editors: W. Rheinboldt and D. Siewiorek.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *18th International Conference on Parallel Processing*, pages 160–169, Aug 1989.

[RK89] Umakishore Ramachandran and M. Yousef Amin Khalidi. An implementation of distributed shared memory. *USENIX Workshop on Experiences in Building Distributed and Multiprocessor Systems*, October 1989.

[RK91] U. Ramachandran and M. Yousef A. Khalidi. An implementation of distributed shared memory. *Software–Practice and Experiance*, 21(5):443–464, May 1991.

[RTY+87] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, 0ct 1987.

[SB90] Michael D. Schroeder and Michael Burrows. Performance of firefly rpc. *ACM Transactions on Compututer Systems*, 8(1):1–17, Feb 1990.

[Sco87] Michael L. Scott. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering*, pages 88–103, Jan 1987.

[SDRC82] John F. Shoch, Yogen K. Dalal, David D. Redell, and Ronald C. Crane. Evolution of the ETHERNET Local Computer Network. *IEEE Computer*, pages 1–27, Aug 1982.

[Wil89] Christopher J. Wilkenloh. RaTP: A transaction support protocol for *ra*. Master's thesis, School of Information and Computer Science, Georgia Institute of Technology, 1989.

[Zay87] E. Zayas. Attacking the process migration bottleneck. In *Proc. Eleventh ACM Symp. on Operating Systems Principles*, pages 13–24, November 1987.

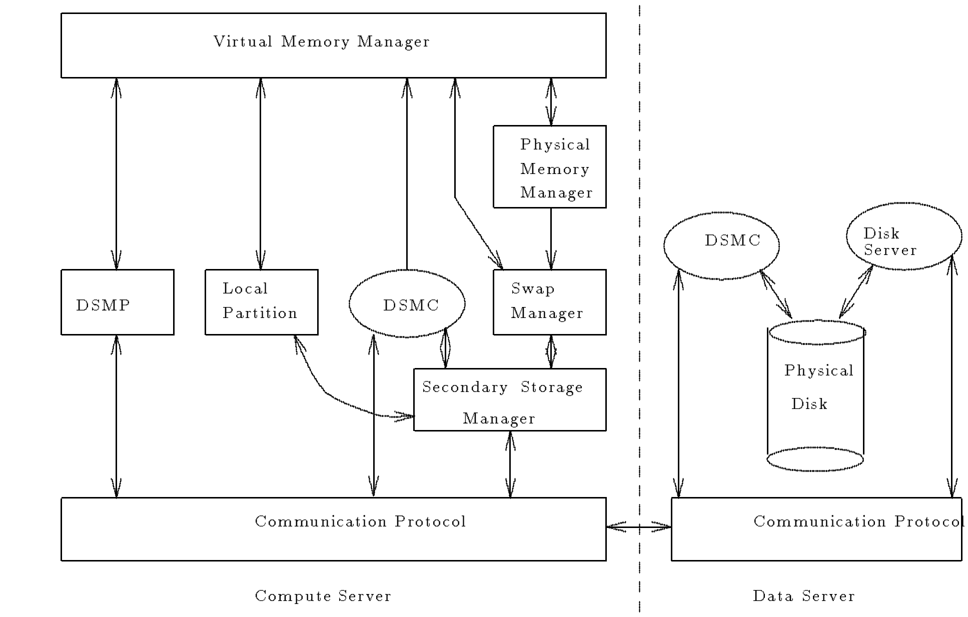

Figure 1: Distributed Memory management architecture

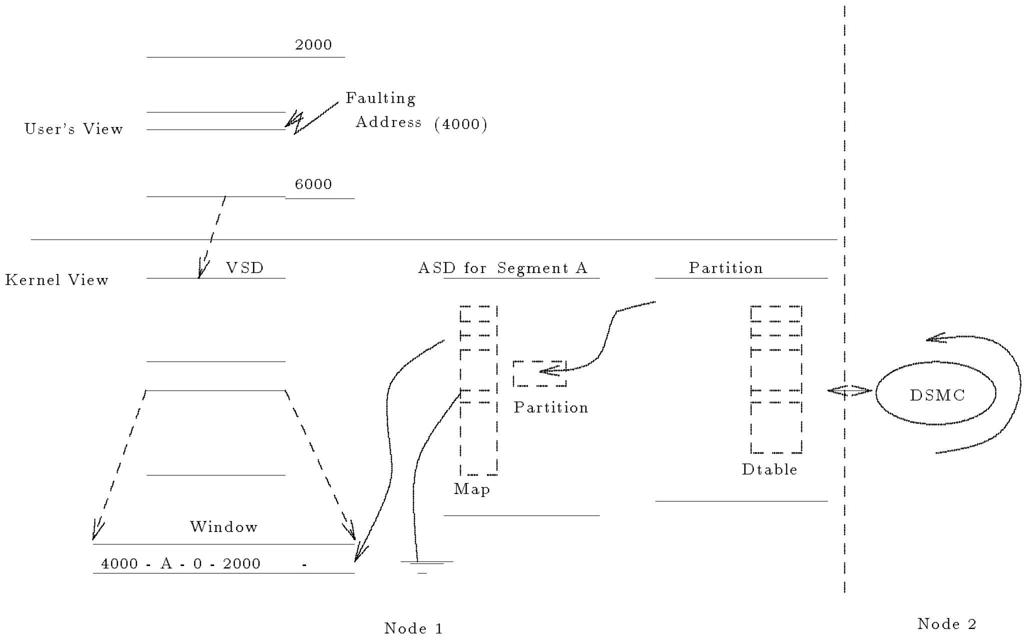

Figure 2: An example of page-fault handling - before the page-fault is serviced

VM Functional Description
Virtual Space calls :
install() — Installs a virtual space in one of the O or P regions.
uninstall() — Inverse of install.
MapPage(VirtualAddress, PhysicalFrame) — Maps a physical frame to a virtual address.
UnMapPage(VirtualAddress) — unmaps the physical frame associated with the virtual address.
WhoMaps(VirtualAddress) — returns the window corresponding to a virtual address.
AddMapping(StartingVirtualAddress, Length, SegmentName, Offset, ProtectionAttribute).
DeleteMapping(StartingVirtualAddress, Length).
Flush() — Flushes the virtual region to partition(s).
VM manager :
GetSegmentPage(SegmentName, PageNo, PhysicalFrame).
PM and Swap Managers :
AllocatePhysicalFrame(), DeallocatePhysicalFrame().
SwapIn(PhysicalFrame), SwapOut(PhysicalFrame).
Partition Interface :
Create(SegmentName), Destroy(SegmentName).
Activate(SegmentName), Deactivate(SegmentName).
ReadSegmentPage(SegmentName, PageNo, PhysicalFrame).
WriteSegmentPage(SegmentName, PageNo, PhysicalFrame).

Table 1: Summary of VM subsystem calls

	Domain	Ivy	Clouds	Mach	Agora	Memnet	Choices	Mether
Virtual Memory	Yes	Yes	Yes	Yes	No	No	Yes	Yes
Granularity	Page	Page	Page	Page	Data structure	32-bytes	Page	Page
Memory Model	SC	SC	SC and User-defined	SC	User-defined	SC	SC	User-defined
Coherence Protocol	Version number based	WI	LBS	WI	WU	WI	WI and LBX	WU
Synchronization	Yes	No	Yes	No	No	No	Yes	No
Dedicated Hardware	No	No	No	No	No	Yes	No	No

Table 2: Comparison of DSM systems

Legend:

SC : Sequential Consistency

WI : Write Invalidate — WU : Write Update

LBX : Lock-based (exclusive) — LBS : Lock-based (exclusive and shared)

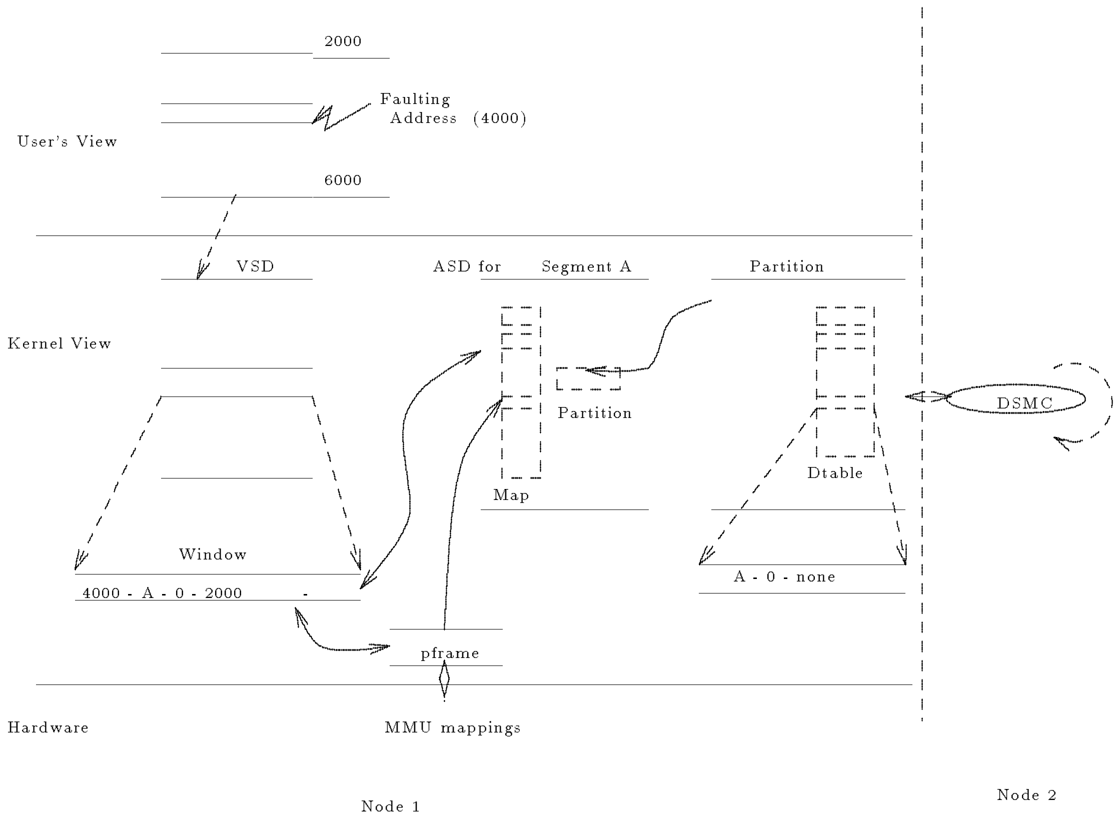

Figure 3: An example of page-fault handling - after the page-fault is serviced

Basic system performance (All times are in milliseconds)	Breakdown	Measured Total Time
Basic context switching		0.150
Null Round trip time using ETHERNET system object		1.590
Transmit processing by sender	0.450	
Wire overhead (64 bytes, computed)	0.051	
1 Context switch at receiver	0.150	
Transmit processing by receiver	0.450	
Wire overhead (64 bytes, computed)	0.051	
1 Context switch at sender	0.150	
TOTAL TIME	1.302	
Null Round-trip time at RaTP level		3.560
Initiating request transaction (including a 32-byte copy of RaTP header into an Ethernet buffer)	0.800	
Wire overhead (64 bytes, computed)	0.051	
RaTP processing at server before wakeup of server thread	0.265	
2 Context switches at the receiver (artifact of the RaTP design)	0.300	
Server processing of the request	0.150	
Initiating reply transaction (including a 32-byte copy of RaTP header into an Ethernet buffer)	0.600	
Wire overhead (64 bytes, computed)	0.051	
RaTP processing before wakeup of the client thread	0.265	
2 Context switches at the sender (artifact of the RaTP design)	0.300	
Client processing of the reply	0.100	
TOTAL TIME	2.882	

Table 3: Basic Timings

Basic system performance (contd.) (All times are in milliseconds)	Breakdown	Measured Total Time
Transfer time at RaTP level (64-byte request one-way, 8K byte other-way)		12.300
Initiating request transaction (including a 32-byte copy of the RaTP header into a Ethernet buffer)	0.800	
Wire overhead (64 bytes, computed)	0.051	
RaTP processing at the server before wakeup of the server thread	0.265	
2 Context switches at the server node	0.300	
Server processing of the request	0.150	
Initiating reply transaction (8K bytes, 6 packets) (including one copy of 8K bytes plus headers into 6 Ethernet buffers)	2.724	
Wire overhead (8K bytes + headers, computed)	6.794	
RaTP processing before wakeup of client thread	0.737	
2 Context switches at the client node	0.300	
Client processing of reply (accepts data in a buffer, no copying involved)	0.100	
TOTAL TIME	12.221	

Table 4: Basic Timings (contd.)

DSM operations (segments in memory) (All times are in milliseconds)	Breakdown	Measured Total Time
Get from a data server (no forwarding)		15.500
Basic RaTP 8K bytes transfer	12.300	
1 Context switch at the server	0.150	
DSM processing at the server (updating state information)	1.600	
One 8K byte copy from Ethernet buffers into a client buffer	1.450	
TOTAL TIME	15.500	
Get from a data server (with forwarding)		18.500
Basic RaTP 8K byte transfer	12.300	
1 Context switch at the server on the owner node	0.150	
DSM processing at the server (updating state information)	0.800	
Sending a forwarding request to the current keeper	1.780	
1 Context switch at the keeper node	0.150	
DSM processing at the keeper (updating state information)	1.600	
One 8K byte copy from Ethernet buffers into a client buffer	1.450	
TOTAL TIME	18.230	

Table 5: DSM Timings

Page Fault Service	Time (in milliseconds)
Segment owned by a remote data server (no forwarding)	16.30
Segment owned by a remote data server (with forwarding)	19.30
For a perishable segment (with zero-fill)	1.52
For a perishable segment (without zero-fill)	0.65

Table 6: DSM and VM interaction

Operation (All times are in milliseconds)	Clouds	Amoeba	Mach	Chorus	Mirage
Accessing an 8K zero filled region	2.15	—	3.12	1.50	—
DSM remote page fetch	15.50 (8K)	—	—	—	27.50 (1K)
servicing a remote page-fault (8K)	16.30	13.10	—	—	—

Table 7: Comparison of performance with other systems