A Distributed Shared Memory Facility for FreeBSD*

Pedro Souto†‡ and Eugene W. Stark§
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794 USA

Abstract

This paper describes the design and implementation of a distributed shared memory facility we have implemented for the FreeBSD operating system (a descendant of 4.4BSD that runs on the PC architecture). Interesting aspects of the design are: (1) the consistency protocol uses unreliable datagram communication, but is robust with respect to message loss, and in the normal case requires only two datagrams to handle a read fault; (2) the facility provides a simple programming interface that does not require any socket or network programming to use; (3) the facility extends the FreeBSD VM system in a very non-intrusive way.

1 Introduction

A distributed shared memory (DSM) facility permits processes running at separate hosts on a network to share virtual memory in a transparent fashion, as if the processes were actually running on a single processor [LH89]. This is accomplished with the help of the virtual memory (VM) subsystem, which identifies page faults on DSM pages and invokes the DSM subsystem to retrieve data over the network and perform necessary synchronization.

This paper describes the design and implementation of a distributed shared memory system we have built for the FreeBSD 2.1 operating system, a descendant of 4.4BSD that runs on the PC architecture. The following goals were important in shaping the design of our facility:

- A simple client application interface, which would be as close as possible to ordinary memory.
- To make the basic DSM read page/write page operations as efficient as possible in the normal case.
- A nice fit with the existing FreeBSD VM system, with minimal changes to existing FreeBSD kernel code, and providing flexibility for experimentation with different consistency protocols.

The programming model presented to client applications centers around the notion of DSM *objects*, which are used in a fashion analogous to the use of memory-mapped files. No network or socket programming is required of an application in order to use the DSM facility. At the kernel level, the notion of DSM object fits together nicely with the VM objects that already exist in the Mach-derived FreeBSD VM subsystem, enabling us to add the DSM facility to FreeBSD in a very non-intrusive fashion.

To achieve efficiency and low communication complexity, we adopted as a basic design decision that unreliable datagram communication (our implementation uses UDP) should be used whenever possible. The protocol we designed, which is a write-invalidate protocol that ensures sequential consistency [Lam79], requires in the normal case only two datagrams (request and reply) to retrieve a copy of a page from a remote host, and a total of $n + c + 1$ datagrams (or an n-way multicast plus $c + 1$ individual datagrams) to obtain write permission on a copy of a page, where n is the number of hosts interested in the object and c is the number of hosts that actually hold copies of the page. The protocol is robust in the face of loss or reordering of datagrams, though in this case or in the case of contention for pages, additional messages may be required. Measurements of basic latencies show that our read page fault are less than 3 ms, which is within 1.5 ms of the best published results [BB93] we know of.

The implementation of the DSM facility required

*Product names used in this publication are used for identification purposes only and may be trademarks of their respective companies or organizations.

†Supported by a PRAXIS XXI fellowship, from the Junta Nacional Científica e Tecnológica of Portugal.

‡E-mail address: `souto@cs.sunysb.edu`

§E-mail address: `stark@cs.sunysb.edu`

only minimal changes to the existing FreeBSD kernel code: only about 100 lines of additions or modifications were made to previously existing kernel files. The rest of the system is split between about 3000 lines of new kernel code and about 5000 lines for a user-level DSM server program. The user-mode server implements essentially all aspects of the consistency protocol, and interacts with the kernel through a narrow interface, thus allowing easy experimentation with various consistency protocols.

The remainder of the paper describes in more detail some of the more interesting aspects of our system.

2 Architectural Overview

2.1 Programming Model

Our DSM facility centers around the concept of a *DSM object*, which is a virtual address space that consists of a sequence of shared pages. A process wishing to access a DSM object must first obtain for that object: (1) a UID, which uniquely identifies that object among all other DSM objects in the world, and (2) the network address of a DSM server that knows about that object. A UID is obtained either by requesting the creation of a new DSM object, or else by receiving the UID of an existing DSM object through some communication channel outside the DSM facility. Network addresses are obtained by similar means. Once a process has the UID of a DSM object, and a corresponding server address it requests to *attach* to the object, using a system call provided for this purpose. After attaching to the object, the process uses another system call to *map* pages from the DSM object into its own virtual address space. When a process has finished accessing a DSM object, it asks to *detach* from the object; in response to this request the DSM facility deletes any existing mappings of that object. The attach/map/detach paradigm for DSM objects is analogous to the open/map/close paradigm for memory-mapped files. It is also quite similar to what is provided by the System V `shm` [ATT90] shared memory facility for interprocess communication.

Once a process has mapped DSM pages into its virtual address space, normal memory references to the mapped virtual addresses are used to access data in the DSM object. As usual, such memory references will cause a *page fault* if either the corresponding page is not resident in physical memory, or else the page does not have the appropriate access permissions set. When a page fault occurs for a virtual address that has been mapped to a DSM object, the kernel page fault handler dispatches a request to the DSM subsystem. The DSM subsystem handles this request, communicating, if necessary, with DSM servers elsewhere in the network either to obtain a copy of the page to be read, or else to synchronize with the other servers to ensure that a write operation can be performed on a page without violating data consistency guarantees. Once the required communication and synchronization has been performed, the DSM subsystem responds to the page fault handler, and the faulting process is allowed to continue.

An important feature of the above programming model is that processes using the DSM feature do not have to contain any code for communication over the network. The only aspect of network programming that shows through the interface is the network address required initially to attach to a DSM object, however, this network address can be treated opaquely, as simply a string of bits that is passed to the kernel as an argument to the *attach* request.

2.2 System Structure and Kernel Interfaces

The DSM subsystem has a client/server structure, and contains both kernel and user-level components. The overall organization is depicted in Figure 1. *Clients* are the user-level application processes that make use of the DSM facility. Arbitrarily many clients can run on a single host computer. To support the DSM operations of the clients, a single DSM *server* process runs on each host computer providing DSM service. The DSM server is also a user-level process, though it is a privileged process that makes use of special DSM system calls provided by the kernel. The kernel portion of the DSM subsystem consists of (1) DSM *pager* code, which runs on behalf of a client process as a result of a page fault, (2) *client system calls*, which allow clients to attach, map, and detach DSM objects as described above, and (3) *server system calls*, which provide the DSM server process with the access to the VM system it needs to carry out its function.

As described above, the kernel page fault handler invokes the DSM pager code in response to a page fault by a client process involving a virtual address that has been mapped to a page in a DSM object. The DSM pager does not itself perform any communication or synchronization with remote DSM servers. Instead, it sends a request datagram to the local DSM server indicating the type of service that is required, and then sleeps awaiting a response. The local DSM server receives and handles this request datagram, possibly communicating with DSM servers elsewhere in the network as a result. When the required communication and synchronization has been performed, and any requested DSM

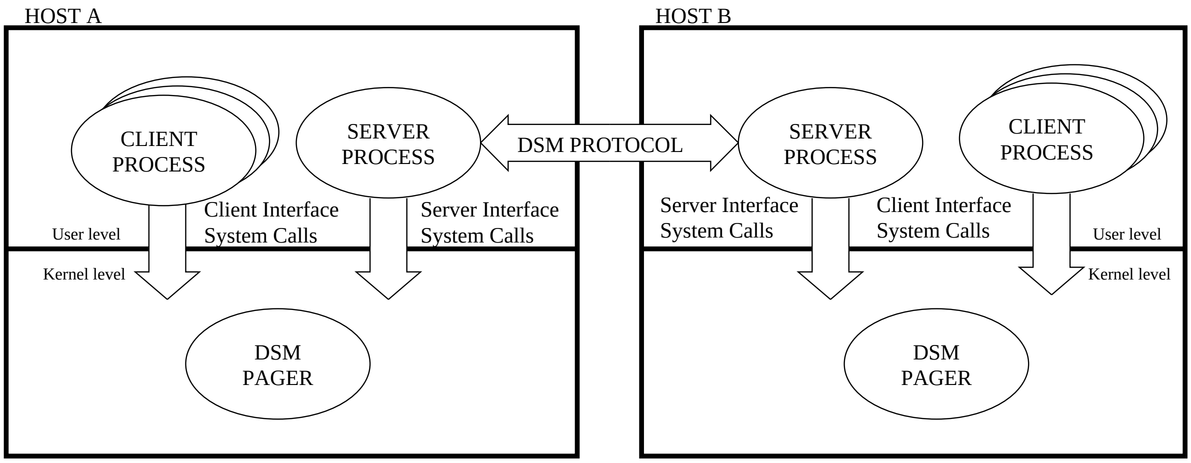

Figure 1: General architecture of the DSM facility.

data is available in local physical memory, the DSM server uses a special system call to awaken the client process sleeping in the DSM pager. The special system call is used so that the DSM server can wake up clients directly in the kernel, instead of requiring every client application to contain code for receiving and interpreting reply datagrams from the server.

To simplify the structure of the DSM server program, no attempt is made by the server to keep track of the status of client operations in progress and to ensure they succeed. Thus, it is possible that a request sent by the local DSM server to a DSM server on a remote host might fail to elicit a response from the remote host; this failure in turn would mean that the local server might never respond to the client that issued the request. In such a situation, the onus is on the client to get things moving again: if the local DSM server fails to awaken the client process after a suitable interval, the client times out from the kernel sleep routine and resubmits the request to the server.

The client system call interface consists of the following system calls (Figure 2a): `dsmcreate`, `dsmattach`, `dsmdetach`, `dsmmap` and `dsmwait`.

The `dsmcreate` call takes as an argument a size in bytes, and causes a new DSM object of that size to be created. The UID of the newly created object is returned. The `dsmattach` call takes as arguments the UID of a DSM object and the network address at which a server who knows about that object can be contacted, and it arranges for the calling process to become attached to the specified DSM object. The return value indicates success or failure. The `dsmdetach` call takes the UID of a DSM object as its single argument, and it causes the calling process to become detached from the specified object. The `dsmcreate`, `dsmattach`, and `dsmdetach` system calls are not actually executed in the context of the client process. Rather, they cause a request datagram to be sent to the local DSM server, who performs the requested service and returns a response to the client waiting in the kernel.

The client `dsmmap` call is identical to that of the previously existing `mmap` call, used to memory map files, except that `dsmmap` requires the UID of a DSM object to be supplied instead of a file descriptor. In spite of the overlap between `dsmmap` and `mmap`, we chose to keep them separate in the current implementation to avoid modifications to existing code. The `dsmwait` call is used by a client process to avoid expensive busy waiting on DSM data. It takes as arguments the UID of a DSM object and the offset of a particular byte in that object, and it causes the caller to sleep as long as it can be guaranteed that the data byte at that offset has not been changed. As soon as this guarantee can no longer be made (for example, if a remote host obtains write permission on the page containing the particular byte), the server awakes the client, which returns to user mode.

The server system call interface consists of the following system calls (see Figure 2b): `dsmservice`, `dsmcreate`, `dsmdelete`, `dsmrespond`, `dsminvalid`, `dsmwritepage`, `dsmsendpage`, and `dsmrecvpage`. The `dsmservice` call is used by the DSM server process on startup to identify itself to the kernel, to prevent any other DSM server processes from starting, and to enable access to the remaining server calls. The `dsmcreate` and `dsmdelete` calls are used to inform the kernel of the creation and deletion of a DSM object. The `dsmrespond` call is used by the server to wake up a client process sleeping in the kernel while awaiting DSM service. The DSM server uses

```
int dsmcreate(int size);
int dsmattach(int objid, struct sockaddr *addr, int len);
int dsmdetach(int objid);
caddr_t dsmmap(caddr_t addr, int len, int prot, int flags, int objid, off_t offset);
int dsmwait(int objid, off_t offset);
```

a) Client system call interface.

```
int dsmservice(int socket);
int dsmcreate(int size);
int dsmdelete(int objid);
int dsmrespond(int responseid, int result);
int dsmwritepage(int objid, off_t offset);
int dsminvalid(int objid, off_t offset);
int dsmsendpage(int objid, off_t offset, dsm_data_packet_t dpkt,
                struct sockaddr *addr, int len);
int dsmrecvpage(int objid, off_t offset);
```

b) Server system call interface.

Figure 2: System call interface.

the `dsmwritepage` call to tell the kernel that it is safe to write enable a particular page of DSM data. The `dsminvalid` call causes the kernel to invalidate any copies it may have of a particular page of DSM data, so that subsequent attempts by clients to access these pages will fault. Finally, the `dsmsendpage` and `dsmrecvpage` are called by the server to send and receive a page of DSM data over the network. System calls are provided for this in order to enable the transmission and reception of DSM data directly from or to the appropriate page of physical memory, so that the user-level server process never touches the actual DSM data. Without these calls, sending a page of DSM data to a remote host would be a much more costly operation involving the copying of data from the kernel to the server process' address space, then copying the data back into the kernel for transmission over the network, followed by the reverse sequence at the destination host.

Note that although the system call interface provides several logically separate system calls, in fact the implementation uses only one actual system entry, `dsmsys()`, which dispatches on its first argument to invoke the appropriate function. This scheme saves system call numbers and is similar to the system call interface of System V shared memory.

In the first version of our system, the DSM server executed completely in the kernel, similar to what occurs in the NFS network file system. This was done for efficiency reasons, and because at the outset we did not have a clear picture of what sort of kernel interface would be required for a user-level server. Unfortunately, the complexity of the server data structures and storage management issues were such that it became too difficult to completely debug a kernel-mode server. In fact, one of the most difficult aspects of the server implementation was implementing a suitable reference counting scheme for DSM data structures, so that DSM resources would be reclaimed automatically when no processes were using them any more. To aid in debugging, we decided to reimplement the server as a user-level process, which communicates with the kernel through the narrow, system call interface just described. This interface is largely independent of the details of the consistency protocol, a feature we have found very useful while refining and debugging our particular protocol.

3 Details of the Kernel DSM Subsystem

An important design goal for our DSM facility was to have it mesh nicely with the structure of the FreeBSD VM system, and to require minimal modifications to existing code. We feel we were reasonably successful at meeting this goal; in the rest of this section we describe in more detail some of the more interesting aspects of the design.

The FreeBSD virtual memory system is based on that of 4.4BSD [MBKQ96], which in turn is derived from that of Mach (Figure 3). A fundamental concept in the Mach VM system [Tev87] is the concept of a VM *object*, which consists essentially of a placeholder for a sequence of physical pages, together with an associated *pager*, which is a set of functions that can be invoked to retrieve data from a backing store,

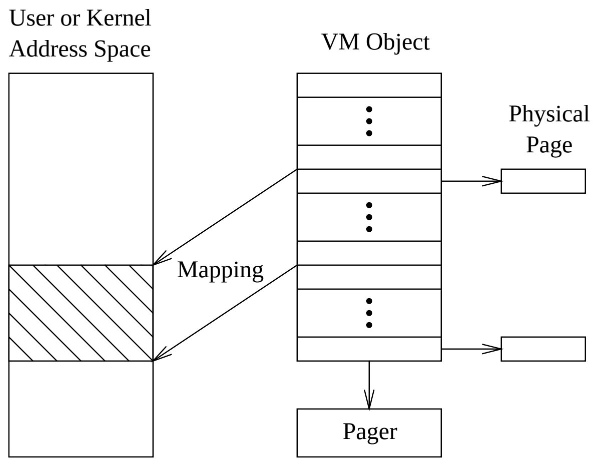

Figure 3: Simplified view of the Mach VM architecture.

such as a swap area, a file, or a hardware device. A process obtains access to the data in a VM object by mapping some or all of its pages into its address space. Typically a process has only a few mappings of VM objects at a time, but actually there is no limit on the number of mappings or the number of VM objects whose pages are mapped.

In the FreeBSD VM system, the allocation of physical memory pages for a VM object is decoupled from the mapping of the object into a process' address space. Physical pages only need to be allocated in a VM object when an attempt is actually made by a process to access data at an offset in a VM object for which no page has yet been allocated. Such an attempt produces a page fault, and the page fault handler not only allocates a physical page for that data, but also invokes the pager to retrieve the data from backing store. The stock FreeBSD VM system supports three different types of pagers: a *swap pager*, which manages the swap area, a *vnode pager*, which is used to page data to and from disk files, and a *device pager*, which is used to page data directly to a hardware device. Additional types of pagers are easily defined.

Our DSM facility was designed to take advantage of the existing FreeBSD VM subsystem. Each DSM object contains an underlying VM object. Mapping a DSM object into a process' address space amounts to simply mapping the underlying VM object. To support the fetching of data over the network, we introduced a new type of pager, called a *DSM pager*. When a fault occurs on a memory address that is mapped to a page in a DSM object, the page fault handler invokes the DSM pager. The DSM pager determines the status of that page by checking data structures maintained by the DSM subsystem, and then requests the local DSM server to perform any communication or synchronization required to bring a copy of the data into local physical memory, and to write enable it, if necessary.

At the rather coarse level of detail of the description so far, the interaction of the DSM facility with the page fault handler seems quite simple. However there are some technical issues that make things a bit more complex than they seem at first. First of all, the stock FreeBSD page fault handler is an extremely complex routine involving many subtle synchronization issues. In order to avoid a difficult debugging task, and to make it easier to track future releases of FreeBSD, we wanted to modify as little of the page fault handler as possible. Some modification to the page fault handler was necessary, because whereas the DSM pager needed to be informed as to whether a read fault or write fault was being handled, the pager interface in FreeBSD did not contain any provision for passing this information to the pager from the page fault handler.

A second technical issue was that the DSM subsystem had to be responsive to requests from the pageout daemon to clean pages of physical memory. The obvious thing for the DSM pager to do when asked to clean a page would be to write it to the swap area. However, I/O to the swap area has to be asynchronous to avoid blocking the pageout daemon, and since there was already an existing swap pager that contained the complicated code necessary to perform this asynchronous I/O, we wanted to make use of it if possible.

A third issue was how the user-level DSM server could arrange for DSM data stored at the local host to be transmitted over the network, even if this data happens to currently reside in the swap area.

To understand how we dealt with the above technical issues, it is necessary for us to describe one more feature of the FreeBSD/Mach VM system. In the FreeBSD VM system, VM objects can be linked together in so-called "shadow chains," which have a special significance to the page fault handler. When a page fault occurs for a page mapped to the first object in a chain, the page fault handler first consults the associated pager (if any) for that object, to try to handle the fault. If the pager for the first object fails to handle the fault, then the page fault handler tries the second object in the chain, and so on. Thus, each object in such a chain serves as a "backing object" for the preceding object, in the sense that if a page is not found in the preceding object, an attempt is made to obtain the page from the next object. In the original literature [Tev87] describing this scheme, each object in a chain is said to be a "shadow" of the next object. However, this terminology has turned out to be very confusing, so we prefer to use the "backing object" terminology instead.

A major purpose of the chains of VM objects in FreeBSD is to support "copy-on-write" for efficient forking. However, for the DSM facility we use these chains in a different way (Figure 4), which we now describe. We have already mentioned that each DSM object has an underlying VM object, which is the actual target of mapping operations by a process. This underlying object has an associated DSM pager, which encapsulates knowledge of how to obtain pages over the network and how to synchronize with the DSM subsystem at remote hosts. In addition, when a DSM object is created, we also create a second VM object that serves as a backing object for the first, so that underlying a DSM object is always a chain of two VM objects. The pager associated with the backing object is not a DSM pager, but rather a swap pager.

When a page fault occurs for an address mapped into a DSM object, the normal operation of the page fault handler is to check the first of the two underlying VM objects to try to obtain the page. When the DSM pager associated with the first object is invoked, it determines: (1) that no copy of the page is available at the local host, or (2) that a copy of the page is locally available, but it might be paged out locally to the swap area. In case (1), the DSM pager sends a datagram to the local DSM server requesting the transfer of a copy of the page to the local host, and it sleeps awaiting the arrival of the page. In case (2), the DSM pager returns a failure indication to the page fault handler, which then moves to the backing object. The page fault handler either finds the requested data already in physical memory mapped from the backing object, or else invokes the swap pager associated with the backing object to bring the data in from the swap area.

The utility of the two-element chain underlying a DSM object becomes evident when one considers how to implement the "pageout" operation of the DSM pager. Rather than having the DSM pager perform a complicated algorithm for asynchronous I/O to the swap area, in response to a "clean" request from the pageout daemon, the DSM pager simply copies the data from the first object in the chain to the corresponding position in the second object. This doesn't immediately free up physical memory, but if there is a high demand for memory, the pageout daemon will eventually ask the swap pager associated with the second object in the chain to clean its page, in which case the data will be written to the swap area using the standard pageout code.

Thus, the two-element chain of VM objects underlying each DSM object permits the DSM system easy access to the normal swap area, with hardly any additional code required. This organization also pays off when the DSM server wishes to transmit to a remote host DSM data that has been paged out locally. As discussed above, transmission of DSM data is accomplished by the special `dsmsendpage` server system call, to minimize the number of copies of a page's data when sending that page to a remote server. This system call simply maps the page of the *first* object in the two-object chain into the kernel address space, and then copies data from that page to the network subsystem. If the page happens not to be resident in physical memory, a page fault will occur, and since the DSM pager does not have the page, the page fault handler will follow the object chain and find the page in the backing object.

Similarly, the reception of DSM data makes use of the special `dsmrecvpage` server system call, to minimize the number of copies of DSM data during reception. This system call temporarily maps the physical page, which was allocated when the page fault first occurred, into the kernel address space and copies the data from the network subsystem to that page.

To support the scheme described above, the kernel needs to have a certain amount of information about the state of the DSM subsystem. In particular, the kernel keeps a data structure for each DSM object that points to the associated VM objects, and keeps track of the location and status (available locally/available remotely, resident/nonresident, write enabled/write protected) of each page in the object. It also keeps a list of pending DSM requests from clients, so that the proper client process can be identified and awakened when the DSM server responds to such a request. The kernel does not need to know anything about the particulars of the DSM protocol or about remote sites participating in the DSM protocol; this is entirely the responsibility of the user-level DSM server process.

4 DSM Protocol

This section describes the DSM protocol executed by our user-level DSM server processes. Essentially, there are two related protocols: (1) a *membership* protocol, which keeps track of hosts that are currently interested in accessing a DSM object, and (2) the *consistency* protocol, which is executed by hosts wishing to read or write pages in DSM objects. The consistency protocol is executed frequently: every

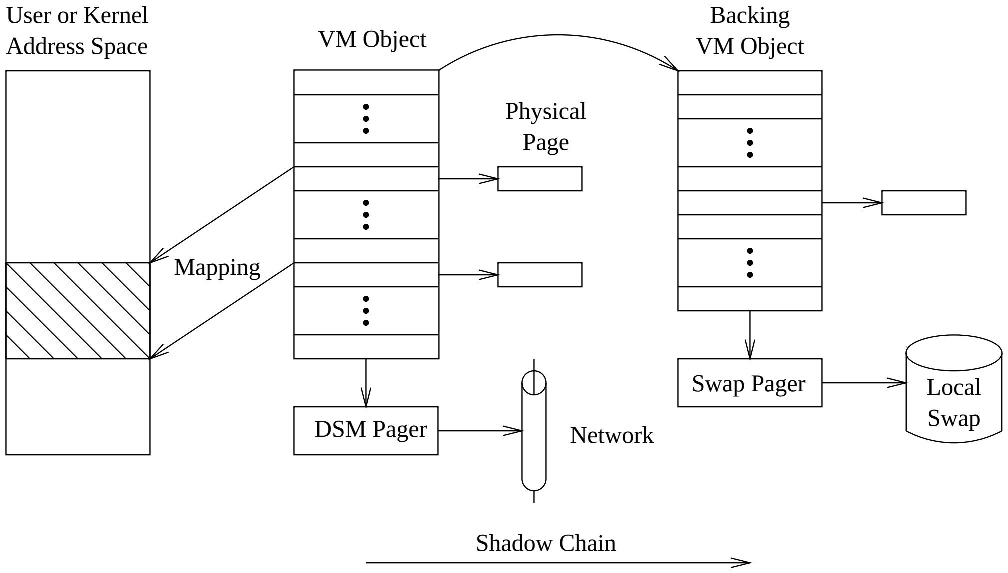

Figure 4: Use of object shadow chains by the DSM facility.

time a host requires an up-to-date copy of a DSM page or needs to obtain write permission on a page. The membership protocol is executed less frequently, but it must be reliable in the sense that the correctness of the consistency protocol depends on the accuracy of the information maintained by the membership protocol. For efficiency, the frequently executed consistency protocol uses unreliable datagrams in all situations but one. On the other hand, the less-frequently executed membership protocol uses reliable datagrams for all communications, where by reliable we mean that a timeout/retransmit scheme is used to guarantee delivery, and ordering of datagrams between each pair of hosts is maintained using a sequence numbering scheme.

To explain the protocol, we first need to introduce some preliminary concepts and terminology. When a DSM object is first created, the only host that knows about that object is the host at which the object is created. This host is called the object *manager*, and it plays a special role in some aspects of the DSM protocol. Application processes at other hosts become informed about the existence of a DSM object by receiving its UID via some form of communication outside the DSM system. The first time an application process at a host tries to attach to the DSM object, the DSM server at that host applies to the manager of the object for *membership* in the object; that is, it asks to be added to the list of all hosts that are currently interested in that object. When the manager grants membership to a new member, it informs all previous members about the new member, and it informs the new member of the current membership list, so that each member of a DSM object knows at all times who all the other members are. Members of an object can resign their membership at any time; in this case a message is sent to the manager, who informs the remaining members about the change.

The consistency protocol belongs to the class of protocols that Li [LH89] calls "dynamic distributed manager algorithms with page invalidation." Just as each DSM object has a manager, each page within a DSM object has an *owner*. However, unlike the object manager, which is fixed at the time the object is created and never changes, the owner of a page changes during execution. Initially, the manager of an object owns all pages in the object. Each time a host receives write permission on a page in an object, it becomes the owner of that page. The owner of a page has the responsibility of safeguarding the data in a page, until it has determined which host will be the next owner and has successfully transferred the data to that host.

Each host that is a member of a DSM object maintains the following state information for the object as a whole: (1) the number of local clients attached to this object; (2) the identity of the object manager; (3) the object UID; (4) the current membership list for the object; (5) the size of the object, in bytes and pages.

In addition, for each page in the object, the following state is maintained: (1) an "owner hint" indicating who the current owner of that page might be; (2) a "version hint" indicating the current version number of the page; (3) a "copies hint" indicating how many copies there might be of the page; (4) a flag indicating whether this host has a copy of the page; (5) a flag indicating whether this host has write permission on the page.

The purpose of the owner hint for a page is to try to route requests for a page quickly to the DSM server that has the most recent data for that page. This hint may become stale, but the protocol guarantees that the host mentioned in the hint is always closer to the actual owner than the host holding the hint. Furthermore, the owner hint is always accurate if the host currently has a copy of the page. The version hint is used to filter out datagrams received out of order, which, if processed, might lead the system to an inconsistent state. The copies hint is an estimate of the number of copies of the page that exist. This estimate is conservative in the sense that the estimate held by the owner of a page is always at least as large as the actual number of copies in existence.

There are two basic operations of the DSM protocol: READ (reading a page) and (WRITE) writing a page. We now discuss these operations in some detail.

4.1 Reading a Page

Figures 5a) to 5c) show three interesting cases of a READ operation. Figure 5a) shows a simple READ, in which the host wishing to obtain a copy of a page has an accurate owner hint, and no messages are lost. In this case, only two messages are required: a READ message from the requesting host to the owner, and a DATA reply from the owner. This is the situation we expect to occur most frequently in actual execution. Note that because all messages exchanged in the consistency protocol use unreliable datagrams, there are no hidden acknowledgments or other messages, and so exactly two messages are required to read a page in this situation.

A slightly more complicated case is when the owner hint held by the requesting host is stale. In this case, the host that receives the READ message uses its own hint to forward the message toward the actual owner, as depicted in Figure 5b). The DATA reply goes directly to the requesting host, who updates its owner hint upon receipt. Since owner hints are updated whenever a host receives new information about the owner of a page, we expect that READ messages will generally be forwarded only a few hops.

Figure 5c) shows a scenario in which a DATA message is lost. In this case, a timeout at the requesting host (by a process sleeping in the DSM pager code) triggers the retransmission of the READ message. This mechanism may lead to the reception of multiple DATA messages containing the same data. To detect this situation, DATA messages include the current version number of the page, and a host receiving a DATA message discards the message if the version is either older than the most recent version of which the host is aware, or the same as the version of any currently held copy. To provide quick error recovery, but to avoid flooding the system with retransmitted READ messages in case of heavy load, we use an exponential backoff scheme with an upper bound to increase the timeout value in case the READ message has to be retransmitted several times.

4.2 Writing a Page

We use a write-invalidate strategy to ensure sequential consistency. That is, before the DSM server at a host allows a client process to modify a page, it invalidates all copies of that page that exist at remote hosts.

In order for a host to initiate a WRITE operation, it is first required to have a copy of the current version of the page. If it wishes to perform a WRITE, but it does not have a current copy, it first executes a READ operation to obtain a copy as described above. There are two reasons for requiring a host wishing to write to have a current copy: (1) it ensures that the host knows the current owner of the page, and (2) it ensures that subsequent message loss during the WRITE operation cannot cause the loss of the data in the page.

Our protocol for writing a page has some uncommon features not usually found in protocols of this type. First, the owner of a page keeps track only of the number of copies of that page, rather than the actual identities of the hosts that have those copies. Second, whereas in all the similar protocols that we know of the write part of the protocol has two distinct operations: the ownership transfer operation and the remote copies invalidation operation, in our protocol the transfer of ownership and the invalidation are combined into a single operation.

Figure 6a) shows what we expect to be the most common case of the WRITE operation, in which there are only a few copies of the page, the owner's copies hint is accurate, and only one host is attempting to write the page. In this situation, the host wishing to write multicasts a WRITE message to every host in the current membership list for the DSM object containing the page to be written. When a member receives the WRITE message, it updates its owner hint to point to the host that issued the WRITE message, and then, if and only if it has a copy of the page, it invalidates that

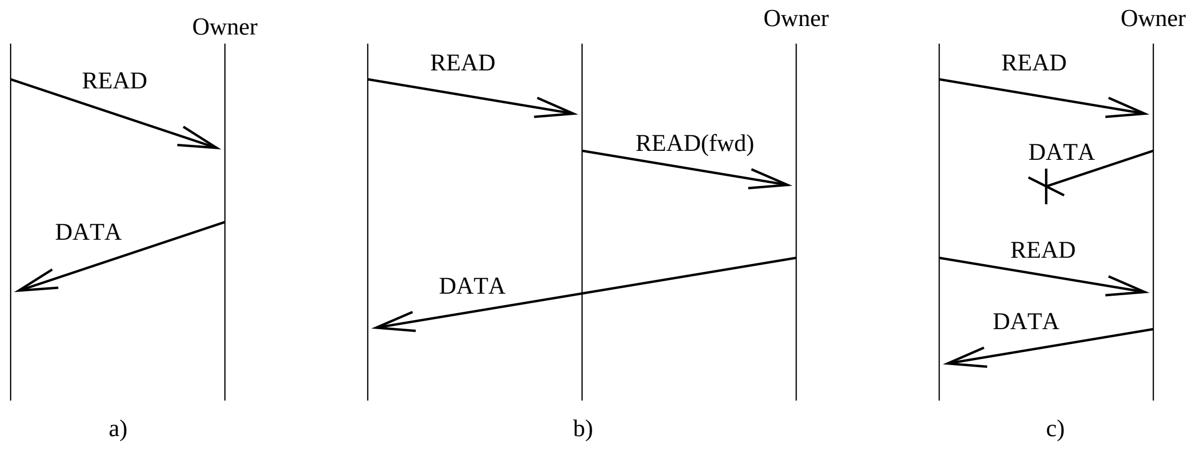

Figure 5: Read operation of the DSM protocol.

copy and responds with a WRITEOK message. The WRITEOK message sent by the current owner of the page implicitly carries with it a transfer of ownership of that page to the requesting host, who, upon receipt of such a message, becomes the new owner, and issues an acknowledgment to the previous owner to release it from any further ownership responsibilities. The WRITEOK message sent by the previous owner also includes a copies hint, which is then used by the new owner to determine when all the remote copies of the page have been invalidated. Specifically, the new owner knows that all copies have been invalidated when the number of WRITEOK messages it has received is equal to the copies hint it received in the WRITEOK message from the previous owner.

As it is absolutely essential that there be no ambiguity about whether ownership has been transferred, the previous owner must wait for its WRITEOK message to be acknowledged, retransmitting the WRITEOK if necessary, before continuing with any other activity regarding this page. The WRITEOK from the previous owner to the new owner is the only reliable datagram used in the consistency protocol. However, observe that the acknowledgement message used by the reliable datagram service is not in the critical path, as the new owner does not have to wait for the acknowledgement to reach the previous owner.

Figure 6b) illustrates what happens in the case of write contention; that is, when two or more hosts try to write the same DSM page at the same time. When the owner of the page processes the first WRITE message for that page, it invalidates its copy of the page and replies with a WRITEOK. If it later sees a WRITE message from some other host for the same version of the page, it simply discards the later WRITE message. This ensures that only one host will become the owner of the next version of the page, and consequently at most one host will be granted the right to write that page.

A host trying to write a page also discards any WRITE messages it receives. This is necessary, because the only alternative would be for the host to invalidate its copy of the page, but then the page would be lost if the host should happen to be granted ownership by the previous owner. Thus, without any special provisions, a deadlock could result when two hosts try to write the same page and each steadfastly refuses to invalidate and send a WRITEOK to the other. To handle this situation, a host starts a timer when it first multicasts a WRITE message. If, by the time the timer has expired, it has received ownership of the page but has not received enough WRITEOK replies, it multicasts a PURGE message to the membership list. Upon receiving a PURGE message, every member is obligated to invalidate any copy it holds, to update its owner hint to point to the sender, and to reply to the sender with a WRITEOK message. If insufficient WRITEOK messages are received after a suitable period, the owner of the page multicasts another PURGE message. This scenario repeats until the owner is sure that all pages have been invalidated.

To be sure that all copies of a page are actually invalidated, the host issuing a PURGE message has to be able to distinguish WRITEOK messages sent in response to the initial WRITE message, and also in response to subsequent PURGE messages. For this purpose, the host maintains a "phase counter," which is an integer variable that is incremented every time the server multicasts a new set of WRITE or PURGE messages. Each such message includes the value of the phase counter, which the recipient

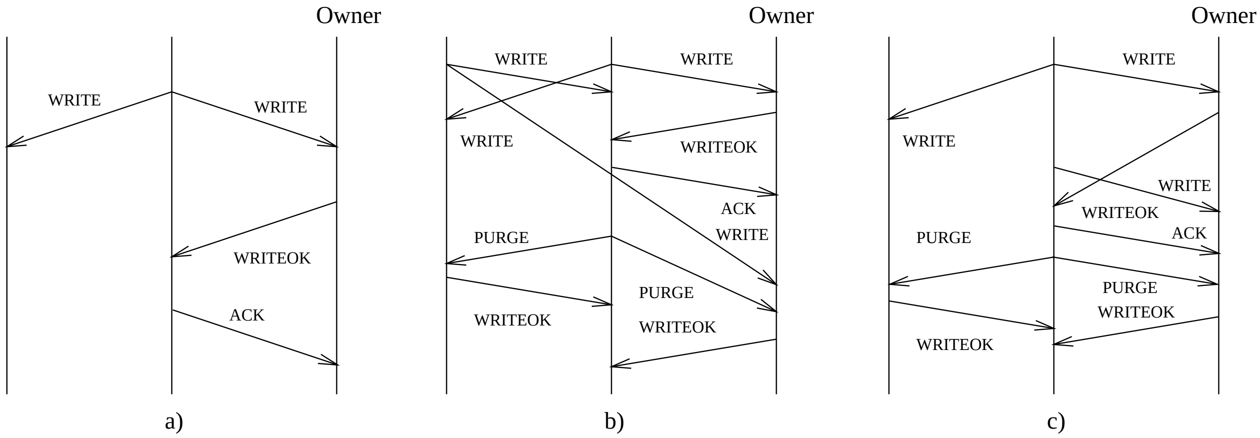

Figure 6: Write operation of the DSM protocol.

echoes back in the WRITEOK response.

Another requirement for correctness of the protocol is that it should not lead to cycles in a page's owner chain. To satisfy that requirement, once a PURGE message has been processed for a version of a page, a host must not again update its owner hint for that page in response to a WRITE message for the same version of that page. Actually, our protocol uses the following stronger policy concerning the updates of owner hints: after processing a WRITE or PURGE message for a version of a page, a host will not again update its owner hint for that page in response to a WRITE message for the same version of that page. However, even if a host has previously received a WRITE message for a version of a page, it *will* still update its owner hint for that page in response to a PURGE message for a version of that page, as long as it is not aware of a more recent version of the page than that specified in the PURGE message.

As mentioned previously, the copies hint held by the owner might not be accurate. For example, due to a slow network or a slow server, a host might retransmit a READ message before it receives the DATA response sent by the owner of the page in response to the original READ message. Since the owner does not keep track of the identity of hosts to which it sent copies of the page, it has no choice upon receiving a retransmitted READ message but to send another DATA message and increment its copies hint to maintain a conservative estimate. This leads to a possibility that a subsequent WRITE operation will deadlock, due to the fact that it will be impossible to obtain enough WRITEOK messages. Eventually, the host wanting to write will time out as described above. If the host has become the owner by the time the timeout occurs, it will multicast a PURGE message as already described. However, there exists the possibility, due to a slow network or slow response from the previous owner, that by the time the timeout occurs, the host wanting to write has not yet become the owner. In this case, that host simply retransmits the WRITE message to the owner of the page, whose identity it knows because the fact that it has a copy of the page means that its owner hint is accurate. Figure 6c) illustrates such a scenario, in which the previous owner has an inaccurate copies hint and the WRITEOK it sends is slow in arriving at the host performing the WRITE. When the retransmitted WRITE message arrives at the previous owner, it discards it because the WRITEOK has already been sent. Eventually, the host performing the WRITE will time out again. By this time, however, it has received the WRITEOK from the previous owner, and thus will proceed to a PURGE operation as above.

In summary, the basic write protocol just described consists of a sequence of phases: an initial WRITE phase, then zero or more phases in which the WRITE message is retransmitted to the owner of the page, then zero or more PURGE phases.

Message loss has basically the same effect for write operations as does uncertainty about the number of copies of a page: the server does not receive enough WRITEOK messages and consequently cannot be sure whether its copy of the page is the only one in the system. Thus, the mechanisms used to handle inaccuracy of the copy hint also handle the loss of messages.

In order to reduce the write time, we use one optimization that is worth mentioning. As shown in Figure 6c), a very slow network or host can trigger a new WRITE/PURGE phase, due to the late arrival of a WRITEOK message at the host performing the WRITE/PURGE operation. If the situation persists, the host performing the WRITE/PURGE operation will never receive enough WRITEOK messages for a phase before its timer for that phase expires. To prevent such a scenario, a host performing a WRITE operation keeps track of the WRITEOK messages received for a fixed number of previous WRITE/PURGE phases, and it accepts WRITEOK messages sent in the scope of any such phase, even though it might have initiated several new phases.

5 Experimental Results and Discussion

We have run reasonably rigorous tests of the consistency protocol using some simple exerciser programs, and we have measured some basic performance parameters. In this section, we describe the results of these tests. Important testing that we have not yet done is to use the system for a realistic application.

In order to evaluate the performance of our implementation, we performed two sets of experiments: one to determine the basic costs of handling read and write faults on DSM pages, and another to assess the scalability of our protocol. In these experiments we used PC's each with either Pentium 75 MHz or Pentium 100 MHz microprocessors, with 256 Kbyte "write back" second level cache and 16 Mbyte of main memory. These machines are interconnected by an 100 Mbps Ethernet via a SMC EtherPower 10/100 adapter, which sits in the PCI bus and supports DMA.

To determine the costs of both read and write faults, we ran a simple "ping-pong" exerciser program in which two clients running on different hosts alternately write a DSM page in such a way that the page bounces back-and-forth between the two, and there is no write contention. Table 1 shows both the minimum values and the average values over 100,000 operations, for both the read and write times, measured at the kernel level on machines with 75 MHz processors. The read times give the actual time taken to handle a read fault by a client process. The write times include only the time taken to execute the invalidation protocol – in general, handling a write fault may also require an initial read operation to obtain a local copy of the page before beginning invalidation.

These results are encouraging, they are within 1.5 ms of the best published values that we know of [BB93], which were measured on a kernel implementation using a network subsystem specially designed for performance that accesses the network interface directly, bypassing the UDP and other communications software layers. Note however, that the values presented in that paper were measured using IBM RISC System/6000 Model 530, which run with a clock frequency of 25 MHZ, interconnected by point-to-point 220 Mbps optical fiber network.

It is worth mentioning that a "ping" (ICMP echo request/response) between the machines we used in our experiments takes on average 0.44 ms, and the messages exchanged in a ping neither traverse the UDP layer nor are they processed at user level. We believe that improved performance would result due to reduced IPC costs, if the DSM server were moved into the kernel.

Table 2 shows the breakdown of both the read and write faults times (again, these are averages over 100,000 operations). The IPC times, RD_IPC and WR_IPC, measure the time from the moment the DSM pager sends a request to the local DSM server to the time that server starts processing that request. Thus it includes the time to send a message to the local host, the time needed for a context switch, the time to receive the message from a socket, the time spent by that message in a queue of messages in the DSM server, and the time required to do some preprocessing of the message. For this experiment, the message queue at the DSM server is empty, so the message is processed immediately. The DSM server times, RD_SRV and WR_SRV, are the times taken by the server to satisfy the DSM pager request; that is, the times taken to get a page from a remote server or to invalidate copies of the page in remote servers.

To assess the scalability of our algorithms we measured the time to invalidate the remote copies of a page under conditions of no contention and when the number of copies of the page is equal to the number of members of the object. As one would hope, the invalidation time depends roughly linearly on the number of remote copies: our measurements showed a constant overhead of 1.3ms, plus an additional 0.4ms for each copy to be invalidated, over the range of 2 to 11 remote copies. Note that our current implementation sends out WRITE messages sequentially. We expect that using a multicast facility for this would decrease the per-copy overhead.

We also ran some experiments under conditions of high write contention. These experiments revealed two potential problems with our current protocol. First, the slower machines (the 75MHz processors) tended to starve in favor of the faster machines (the 100MHz processors). The second problem, which was exacerbated by the first, is that the simple time-out/retransmit policy with exponential backoff we currently use to handle message loss and deadlock

	Read	Write
minimum time (ms)	2.7	2
average time (over 100,000 values) (ms)	2.9	2.2

Table 1: Basic read and write times measured at kernel level.

	RD_IPC	RD_SRV	WR_IPC	WR_SRV
average time (ms)	0.58	2.1	0.57	1.4

Table 2: Breakdown of the read and write times shown in Table 1.

situations did not adapt well to varying loads, resulting in a large number of retransmitted messages under conditions of high contention. We are considering ways of improving the timeout heuristics, and of modifying the protocol to alleviate the starvation problem.

A basic aspect of our design that we did not evaluate experimentally was our decision to use UDP rather than TCP for the consistency protocol. We continue to feel that any simplifications in the protocol that might be afforded by the use of TCP as an underlying reliable communications protocol would be more than offset by the overhead of additional acknowledgements, the loss of control over the retransmission policy, and the need for an additional software layer to re-implement a message-based communication model on top of the stream-based TCP protocol. In addition, the use of TCP would not allow us to take advantage of multicast support provided by IP. In spite of the above, to validate our belief in the superiority of datagrams over streams as an underlying protocol, it would probably be worthwhile to perform some experiments in which we compare the performance of the UDP-based version of our protocol with a reasonably similar TCP-based version.

Another interesting question concerns the impact on paging performance of our scheme for "pageout" of DSM pages by copying the data to the second object in the shadow chain. It is possible that the "second chance" this scheme gives to pages containing DSM data could have unforseen interactions with the pre-existing page replacement policy. To examine these questions, we would have to test our system with a realistic application, under conditions that would cause heavy pageout to the disk. We have not yet performed such tests.

6 Related Work

Research in the area of DSM systems has been very intense and there is an extensive literature [Esk96]. We compare our system to other software DSM implementations that support a sequential consistency model. The main points that we feel distinguish our facility are: (1) The consistency protocol is a lightweight, distributed protocol, which uses unreliable datagrams, but which is robust with respect to message loss, reordering, or duplication. (2) The facility is for a version of Unix (FreeBSD 2.1) for which source code is readily available and which runs on commodity hardware. (3) The clean interface between the user-level server and the kernel should facilitate experimentation with a variety of DSM protocols.

Most of the first implementations of software DSM systems, including that of IVY [Li88], the DSM system for Clouds [RK89] and Mirage [FP89], were implemented in operating systems different from Unix and the consistency protocols used assumed reliable communications. Mether [MF89] is the exception among early implementations. It is a kernel level implementation of DSM for SunOS 4.0. Although it uses UDP, it relies on HW support for error correction. A more recent implementation of Mirage [FHJ94], although for the AIX operating system, uses reliable communication services. Furthermore, every page request has to be sent to the page's manager, which sends it to the current owner of the page.

The DSM systems described in [FBS89] and [AAO92] take advantage of the VM external pager interface provided by Mach and CHORUS microkernels, respectively. The consistency protocol of Mach's DSM uses only a point-to-point reliable communication service, in contrast to ours which uses multicast and unreliable communication services. Chorus's DSM uses one of Li's dynamic manager distributed algorithms with page invalidation [LH89] but the authors do not specify which and their description of the protocol is rather incomplete. In addition, they do not provide details with respect to the kind of communication services used for IPC. As does our DSM system, Chorus' DSM supports paging out pages to disk, but, in contrast to our system, paging out is handled by the object manager.

Both DVSM6K [BB93], a DSM system developed for AIX v3, and the DSM system developed for the TOPSY multicomputer [SWS92] have an architecture very similar to that of our system. However, the latter was designed for a distributed memory multiprocessor system using a multiprocessor operating system, and DVSM6K assumes that the communication system provides reliable communication, *i.e.* in-order delivery, no message loss and no data corruption.

7 Conclusion and Future Work

In this paper we described a DSM facility, supporting a sequential consistency model, for FreeBSD, a freely and widely available version of Unix. We believe that this facility meets most of our design goals. The consistency protocol is a lightweight protocol that uses only UDP/IP, but is nevertheless tolerant to both message reordering and message loss. We were able to define a very simple client application interface based on the Unix `mmap()` interface. One of the most successful aspects of our design is its smooth integration into the VM subsystem of FreeBSD, which required very little in the way of modifications to existing code. We believe that it should be possible, with minimal effort, to port this code to other Unix systems, such as OSF/1, with Mach-based VM subsystems.

Besides improving the performance of our system in ways that have already been discussed, we are interested in using the facility for real applications. We are especially interested in the idea of using DSM as a tool for programming distributed applications, rather than for concurrent computation, which has been the focus of most DSM research.

8 System Availability

We are making our code available to anyone interested under a Berkeley-style copyright and license. The code may be obtained via the URL: `http://www.cs.sunysb.edu/~stark/`, or by mailing to one of the authors.

9 Acknowledgements

We wish to thank Professor Tzi-cker Chiueh for making his laboratory facilities available to us, as well as the other members of the Experimental Computer Systems Laboratory for their generous cooperation in the sharing of these facilities.

References

[AAO92] V. Abrosimov, F. Armand, and M.I. Ortega. A Distributed Consistency Server for the CHORUS System. In *Proc. of the Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, pages 129–148. USENIX, March 1992.

[ATT90] ATT. *UNIX SYSTEM V Release 4 - Programmers Guide: System Services and Application Packaging Tools*. Unix Press, 1990.

[BB93] Marion L. Blount and Maria Butrico. DSVM6K: Distributed Shared Virtual Memory on the RISC System/6000. In *Proc. of the 38th IEEE International Computer Conference (COMPCOM Spring 93)*, pages 491–500. IEEE, February 1993.

[Esk96] M. Rasit Eskicioglu. A Comprehensive Bibliography of Distributed Shared Memory. *Operating Systems Review*, 30(1):71–96, January 1996.

[FBS89] A. Forin, J. Barrera, and R. Sanzi. The Shared Memory Server. In *Proc. of the Winter 1989 USENIX Conference*, pages 229–243. USENIX, January 1989.

[FHJ94] B. D. Fleisch, R.L. Hyde, and N. C. Juul. MIRAGE+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers. *Software - Practice and Experience*, 10(24):887–909, October 1994.

[FP89] B. D. Fleisch and G. J. Popek. Mirage: A Coherent Distributed Shared Memory Design. In *Proc. of 12th ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 211–223, December 1989.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690–691, September 1979.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li88] Kai Li. IVY: A shared virtual memory system for parallel computing. In *Proc. of the 1988 International Conference on Parallel Processing*, pages 94–101, August 1988.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, 1996.

[MF89] Ronald G. Minnich and David J. Farber. The Mether System: Distributed Shared Memory for SunOS 4.0. In *Proc. of the Summer 1989 USENIX Conference*, pages 51–60. USENIX, June 1989.

[RK89] U. Ramachandran and M. Y. A. Khalidi. An Implementation of Distributed Shared Memory. In *Proc. of the Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 21–38. USENIX, October 1989.

[SWS92] T. Stiemerling, T. Wilkinson, and A. Saulsbury. Implementing DVSM on the TOPSY Multicomputer. In *Proc. of the Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III)*, pages 263–279. USENIX, March 1992.

[Tev87] Avadis Tevanian. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments.* PhD thesis, Department of Computer Science, Carnegie Mellon University, December 1987.