1992

Remote Memory Backing Storage for Distributed Virtual Memory Operating Systems (Thesis)

James Griffoen

Report Number:

92-052

REMOTE MEMORY BACKING STORAGE FOR DISTRIBUTED VIRTUAL MEMORY OPERATING SYSTEMS

James Griffioen

CSD-TR-92-052
August 1991

Remote Memory Backing Storage for Distributed Virtual Memory Operating Systems *

James Griffioen

Department of Computer Science
Purdue University
West Lafayette, IN 47906
email: jng@cs.purdue.edu

Abstract

The virtual memory abstraction aids in the design and implementation of portable applications. Virtual memory allows applications to execute in an arbitrarily large memory space, independent from the physical memory size of the underlying machine.

Conventional virtual memory operating systems use magnetic disks for backing storage. Magnetic disks provide high data transfer rates, large storage capacity, and the ability to randomly access data, making them an appealing backing storage medium. However, the average seek time of a magnetic disk is several orders of magnitude slower than memory access times. Recent advances in CPU speeds, network bandwidth, and memory sizes have made new types of backing storage with improved performance and greater functionality feasible.

This thesis investigates a new model for virtual memory in which dedicated, large-memory machines provide backing storage to virtual memory systems executing on a set of client machines in a distributed environment. Dedicated *memory servers* provide clients with a large, shared memory resource. Each memory server machine is capable of supporting heterogeneous client machines executing a wide variety of operating systems. Clients that exceed the capacity of their local memory access remote memory servers across a high-speed network to obtain additional storage space. Clients use a highly efficient, special

*This paper was originally published as: James Griffioen, *Remote Memory Backing Storage for Distributed Virtual Memory Operating Systems*, PhD thesis, Department of Computer Science, Purdue University, West Lafayette, IN, August 1991. This technical report was created in response to a sudden (and somewhat delayed) barrage of requests for the original thesis. Please use the original thesis citation when citing this work.

purpose, reliable, data streaming, network architecture independent communication protocol to transfer data to and from the memory server. To reduce the delay associated with accessing remote memory, memory servers use efficient algorithms and data structures to retrieve data, on average, in constant time. In addition to providing a highly-efficient backing store, the model allows data sharing between clients and improves file system performance by offloading the file server.

This thesis also describes the design and implementation of a prototype system. Measurements obtained from the prototype implementation clearly demonstrate the viability of systems based on the model. The prototype shows that remote memory systems offer performance competitive with, and in some cases better than, existing virtual memory systems. Moreover, rapid advances in network bandwidth, CPU speeds, and memory sizes make the model an attractive basis for the design of future distributed systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

TRADEMARKS

1. INTRODUCTION

Virtual memory operating systems have had a significant impact on the design and implementation of computer software. Computer hardware originally provided no support for virtual memory. Both the operating system and user programs executed in a single, unprotected, fixed size address space [Dij68, CMDD62, Org72, RT74]. As a result, programming was an art in which programmers spent large amounts of time and effort reducing the size of their programs to stay within the memory space constraints imposed by the hardware (e.g., see [Knu69, Den70]). Even the operating system conserved memory to maximize the amount of memory available for user programs [RT74, Com84].

Advances in computer technology over the past two decades have revolutionized computer programming and operating system design. Today, most conventional computer architectures, including personal computers, provide hardware support for virtual memory. The hardware assists the operating system in creating independent, protected, virtual address spaces, and provides support for detecting illegal memory accesses. The hardware allows the operating system to map portions of an address space to physical memory and other portions of the address space to backing storage. In the past, magnetic disks have served as the backing store of choice. However, recent technology advances in CPU speeds, network bandwidth, and memory sizes have made new types of backing store feasible.

This thesis proposes and investigates a new model for designing distributed systems in which dedicated, large memory machines provide high-speed backing storage to virtual memory systems executing on a set of distributed client machines.

This chapter reviews the design of conventional virtual memory operating systems, discusses conventional backing storage, and introduces the concept of *remote memory backing storage*.

1.1 Virtual Memory Operating Systems

Operating systems that provide support for virtual memory use the virtual memory support provided by the hardware to create large *virtual address spaces* in which user applications execute. A virtual address space is a logical address space, independent from the machine's physical memory space. The virtual memory abstraction allows programmers to design and implement applications that do not depend on the size of the physical memory of the underlying machine. Programmers are not constrained by a machine-dependent physical memory size because each application executes in an arbitrarily large virtual address space.[1] Virtual memory simplifies program design by allowing programmers to focus on problem solving instead of machine-dependent constraints. Virtual memory allows the size of a program to exceed the size of the physical memory. In addition, the virtual memory system can place each application in a separate address space, protected from all other applications. Protected address spaces eliminate elusive bugs that arise as a result of errors in programs that reference memory locations outside their region [Den70, Den80].

In a virtual memory, applications use *virtual addresses* to identify logical memory locations. Virtual addresses are not necessarily bound to physical memory addresses. Instead, virtual addresses are bound to physical addresses dynamically at run-time. A virtual memory operating system[2] creates the illusion of an arbitrarily large virtual address space by mapping a small portion of the virtual space to physical storage. The physical storage space consists of *main memory* and *backing storage*. Main memory

[1]Although they are much less common, there exist a few systems that support small, fixed-size, virtual address spaces (e.g., smaller than the size of the physical memory); however, these systems exhibit the same protection/sharing properties as systems that support large virtual address spaces.

[2]Throughout this thesis we will refer to operating systems that provide virtual memory support as *virtual memory operating systems*.

refers to the high-speed memory physically located on the machine.[3] Backing storage refers to some type of secondary storage media, typically having access times slower than those of main memory. The operating system maps certain regions of the virtual space to main memory and maps the remaining virtual regions to the backing store. When the user applications exhaust all available main memory, the operating system moves data from main memory to the backing store. Later, when the data is needed, the virtual memory system retrieves the data from the backing store.

1.1.1 Segments and Pages

Virtual memory operating systems typically organize memory into *segments* or *pages* [Den70]. A segment is a variable size memory region consisting of a contiguous set of memory locations. The programmer or the operating system defines the segments that comprise an application. Each segment is defined by a (segment number, segment length) pair and has a set of protection values that permit or prohibit access to the data contained in the segment.

Segments mirror the typical organization of an application. An application usually contains several distinct regions, each having a particular purpose. For example, an application often contains one or more text regions (e.g., user code or shared libraries), one or more regions for data structures, and a stack region. In a segmented architecture, each of these regions corresponds to a segment. Multiple virtual address spaces may share a segment, and each segment has a set of protection values to prohibit unauthorized access. Because applications are often organized in a modular fashion, segments provide an attractive method for organizing the memory space.

Virtual addresses in a segmented system consist of a *segment number* and an *offset* within segment. The operating system typically uses an *address translation table* to map virtual addresses to physical memory locations. In a segmented system, the address translation table stores the base address of each segment (i.e., where the segment resides in physical memory) and the length of the segment. When translating

[3]Main memory is also referred to as *primary memory*.

a virtual memory address to a physical memory address, the system extracts the segment number from the virtual address, locates the corresponding segment entry in the address translation table, verifies that the offset does not exceed the segment length, and adds the base address to the offset to obtain the desired physical memory address.

The difficulty with segments is that a contiguous set of physical memory locations large enough to hold the segment must be located before a segment can be loaded into memory. A special case of segmentation, call *paging*, eliminates the problem of locating a contiguous memory region large enough to store the segment.

A *page* is a fixed-size segment. The system divides the virtual memory space into fixed-size segments called *pages*, and divides the physical memory space into fixed-size segments called *frames*. In a paged system all segments have the same size. Because the segment size is fixed, the system can store a segment (page) in any frame. Consequently, loading a page from the backing store simply requires locating an unused frame in which to store the page.

Virtual addresses in a paged system consist of a *page number* and an *offset* within page. In a paged system, the operating system uses an address translation table called a *page table* to map virtual addresses to physical addresses (either in primary memory or on the backing storage) as shown in Figure 1.1.

Segmentation and paging can be combined to reduce the overhead associated with virtual memory (e.g., large address translation tables or long memory allocation delays). *Segmented paging* and *paged segmentation*, as described in [Den70] and [PS85], combine segmentation and paging to capitalize on the advantages of both systems.

1.1.2 Memory Management Policies

A virtual memory system uses three basic policies to govern the storage of data: a *placement policy*, *fetch policy*, and *replacement policy* [Den70]. A placement policy specifies the location in memory where data should be placed. A fetch policy determines when data should be loaded from the backing store. A replacement policy

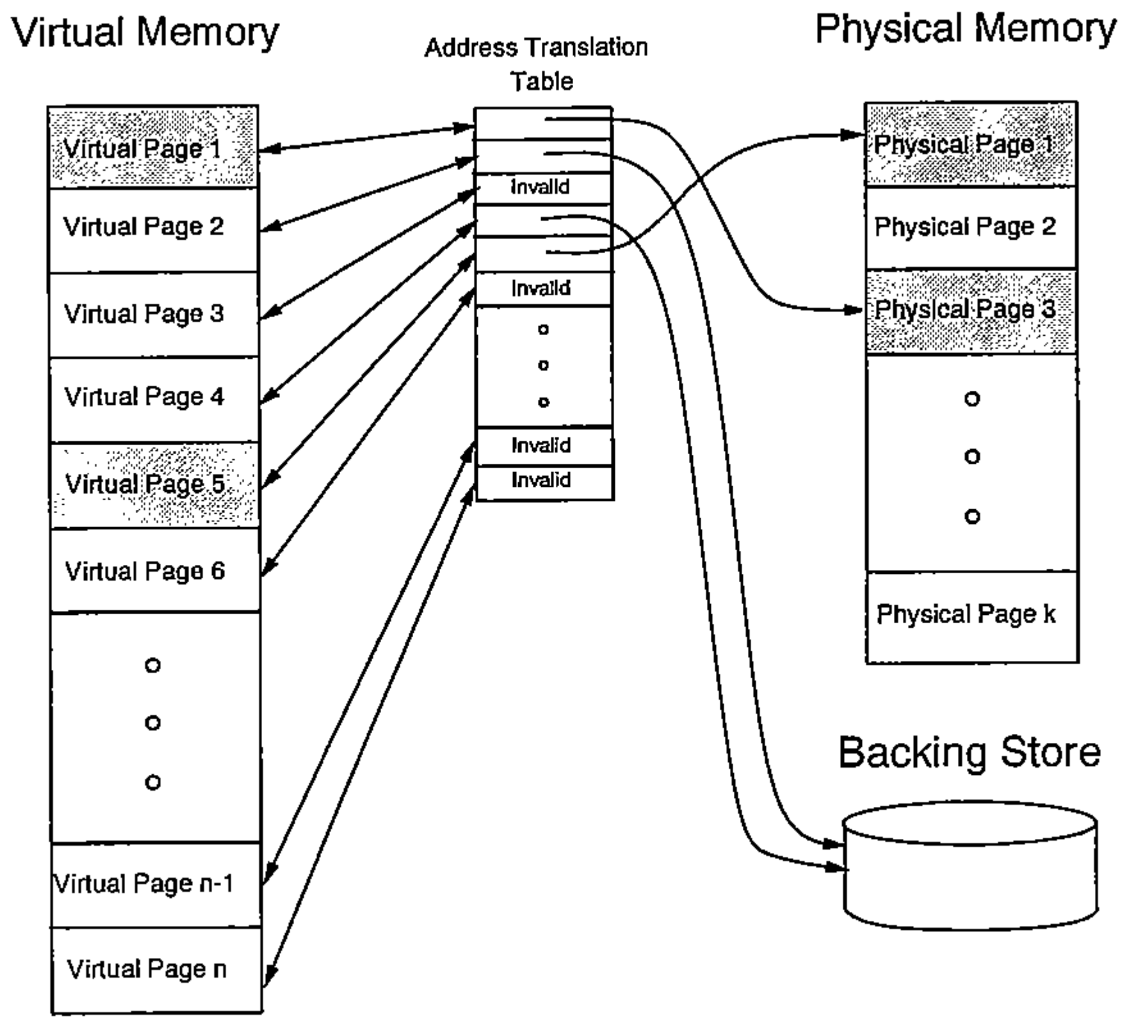

Figure 1.1 Virtual to physical address translation. The operating system maintains an address translation table for each virtual address space. The address translation table contains exactly one entry for every virtual page and specifies the true location of the data (i.e., physical memory or backing storage).

determines which segments/pages should be removed from memory and placed on the backing store. Determining the optimal set of policies for a particular system is a difficult task, and much work has been devoted to the evaluation of various virtual memory policies [Bel66, Den70, ADU71, Den80].

In a paged system, the division of physical memory into uniform frames results in a trivial placement policy. The virtual memory system simply chooses any frame that is available. Moreover, because the operating system typically does not know which pages an application will require in the future, most paged virtual memory systems use a fetch policy called *demand paging.* Under demand paging, the virtual memory system does not retrieve data from the backing store until the data is needed. Although many paged virtual memory systems use the simplistic placement and fetch policies mentioned above, page systems use.a wide variety of replacement policies.

The *replacement policy* defines the rules used to replace data stored in the physical memory. Replacement policies specify the characteristics of segments/pages that should be removed from the physical memory and stored on the backing store. For example, a replacement policy might target the least-recently-used page, the oldest page (first page allocated), or the least-frequently-used page as the page to replace next. A *replacement algorithm* implements the replacement policy. Because most hardware does not supply the information needed to implement the replacement policy directly (e.g., most hardware does not provide the support required to identify the least-recently-used page), replacement algorithms typically approximate the replacement policy.

Replacement algorithms can be classified as *global* or *local* replacement algorithms. Global replacement algorithms choose the next frame to replace from a global set of frames (i.e., all the frames in the system). Local replacement algorithms allocate a fixed number of frames to each application. When an application needs another frame, a local replacement algorithm chooses a frame from the local set of frames allocated to the application.

Example global replacement algorithms include *Least Recently Used* (LRU), *Least Frequently Used* (LFU), and *Most Frequently Used* (MFU). Unfortunately, these algorithms require special hardware support to keep a reference count or timestamp for every physical page.[4] Two common global replacement algorithms, the *Clock* algorithm and the *First-In-First-Out* algorithm, do not require special hardware support, which makes them more attractive than LRU, LFU, and MFU [Bel66, Den70, PS85, Tan87, LKKQ89]. Local replacement algorithms include *Working Set Replacement*, *Page Fault Frequency*, and *Working Size* [Den80, Fin88]. In addition, most global replacement algorithms can double as local replacement algorithms.

1.1.3 Virtual Memory Techniques

The physical memory management component of a virtual memory system provides the mechanisms required to transfer data from primary memory to the backing store and from the backing store to primary memory. Three common virtual memory techniques for transferring data between primary memory and the backing store are:

Overlays: are segments of an application's memory space that are never accessed simultaneously. The application writer knows the structure of the program and defines the overlays. The application assumes the responsibility of transferring the overlays to and from the physical memory at runtime. Overlays require no operating system support or modifications to the operating system.

Swapping: temporarily removes the entire contents of an application's virtual memory space and places it on the backing store. When the application resumes, the operating system must *swap* the entire virtual memory space into memory. If the kernel has long-term scheduling information available, it can quickly free up large amounts of physical memory by swapping out applications that will remain blocked for a substantial amount of time.

[4]We use the phrase *special hardware support* to refer to hardware support not provided by many, if any, conventional architectures.

Paging: temporarily removes individual segments/pages of an application's virtual memory space rather than the entire memory space. The virtual memory system transfers certain segments/pages from the application's memory space to the backing store and then retrieves them before, or when, the application attempts to access them.

Overlays are often considered a virtual memory technique although they do not require that an application execute in a virtual address space. Consequently, overlays are especially useful for hardware that does not support virtual memory. However, the burden of physical memory management falls on the application writer who must carefully analyze the program's behavior, determine the segments of the application that can be overlaid, and write the code to transfer overlays to and from the backing store. Swapping, like overlays, can be used on hardware that does not provide virtual memory support. Swapping allows the operating system to quickly reclaim large amounts of memory by grouping multiple segments or pages together (e.g., all the pages of a process' address space) and transferring them all to backing storage at once. Unfortunately, in many cases the removal of entire processes effectively reduces the level of multi-programming. Paging reclaims memory without reducing the level of multi-programming [PS85]. However, paging typically requires hardware support for virtual memory. Because the operating system does not know in advance which pages an application will require, the operating system relies on the hardware to inform the virtual memory system of accesses to non-resident pages.

The simplicity and uniform treatment of memory[5] in a paged system have made paged systems very popular. Many conventional hardware architectures and virtual memory operating systems provide support for paged virtual memory systems. Consequently, the remainder of this thesis will focus on paged systems although many of the ideas and results presented in this thesis can be applied to segmented systems as well.

[5]Uniform treatment of memory refers to the uniform nature of equal size segments. See [Den70].

1.1.4 Virtual Memory Hardware Support

Most conventional computer architectures provide hardware support for virtual memory; however, memory management hardware differs substantially from one machine to another. For example, the Digital Equipment Corporation VAX architecture uses three distinct page tables to map a virtual address space [Dig85]. Each page table maps a single contiguous virtual region of an address space, limiting each address space to a maximum of three virtual segments.

The Sun Microsystems, Inc. Sun 3 architecture uses a special purpose *memory management unit* (MMU) chip to support virtual memory [Sun86]. The MMU chip contains a fixed-size, high-speed memory that caches a fixed number of page table entries. The operating system must store all other page table entries in the host's memory. Unlike the VAX, the Sun 3 MMU chip can map an arbitrary number of virtual regions in an address space, resulting in large, sparse, virtual address spaces.

The IBM RT/PC and the IBM RS/6000 also support large, sparse, address spaces, but use inverted page tables to reduce the amount of memory allocated for page table storage [CM88, IBM90b, IBM90a]. The hardware uses a hashing function to locate virtual to physical mappings in the inverted page table when performing address translations.

The MIPS chip, used in the Digital Equipment Corporation DECstation series, does not support page tables [Kan89]. Instead, it uses a small, fixed-size translation lookaside buffer (TLB) containing (process id, virtual page, physical page) triples, which it uses to translate virtual addresses to physical addresses.

Clearly, existing virtual memory hardware varies significantly across vendors. Despite the differences, the basic functionality provided by each architecture is the same. In particular, all the architectures provide hardware address translation and allow programs to use virtual addresses rather than physical addresses when accessing instructions and data.

1.2 Conventional Backing Storage

Once a page has been selected for replacement, the virtual memory system moves the page to the backing store. Early virtual memory systems implemented overlays and used a swapping drum for backing storage [Dij68]. Drums provided stable storage and supported random access.

Today, most virtual memory operating systems use magnetic disks for backing storage [PS85]. Magnetic disks provide high data transfer rates, large storage capacity, and the ability to randomly access data, which makes them an appealing backing storage medium. The operating system usually reserves a fixed-size, contiguous region of the disk for backing storage and writes blocks of data directly to the reserved region (i.e., no additional file structure or other organizational structure is imposed on the raw storage provided by the disk) [LL82, Bac86, LKKQ89].

Although magnetic disks support random access, the average retrieval time for a magnetic disk is several orders of magnitude slower than that of physical memory [Hab89, Hag89]. The majority of the disk access time is consumed by the *seek* operation which positions the disk arm over the desired data. Consequently, operating systems often attempt to minimize the number of seek operations. Moreover, depending on the disk device, the virtual memory system may use a complex placement policy to optimize for large data transfers to or from the disk.

More recent virtual memory systems have added a level of abstraction to the paging paradigm [GMS88, OCD$^+$87]. These systems use the file abstraction to hide the underlying storage device from the virtual memory system and allow the operating system to store data on the disk using high-level file operations. The virtual memory system does not need to know the characteristics or organization of the underlying disk device because the file system handles the storage of data on the disk. Some operating systems provide support for a distributed file system and allow diskless machines to use remote files for backing storage [GMS88, Nel86, Wel86].

Unfortunately, using files for backing storage increases the overhead associated with paging. Writing data to a file usually requires a minimum of 2 disk accesses (1 or more to write the data and 1 to update the directory structure), whereas writing directly to the disk requires only 1 access [PS85, Bac86, LKKQ89]. Moreover, file systems often attempt to improve performance with techniques such as *read-ahead*. Read-ahead assumes that most programs access files sequentially and attempts to prefetch additional data whenever the system reads from a file. However, when applied to random access paging activity, prefetching wastes valuable buffer space, degrading both paging and file system performance.

1.3 Remote Memory Backing Storage

Current technology trends provide the impetus to re-evaluate conventional backing storage. In particular, CPU speeds, network bandwidth, and memory sizes are increasing at a rapid pace, and the future promises more advances in these areas [Ous90, Par90]. These changes have made new forms of backing storage with added functionality and competitive performance (i.e., competitive with conventional disks whose speeds have remained relatively constant [Hab89, Ous90]) feasible. Current trends in memory technology create the possibility of machines with very large memories. We envision a distributed system with enough physical memory to back much, if not all, of the virtual memory being used by applications throughout the system. In this new model for virtual memory, dedicated, large-memory machines provide high-speed *remote memory backing storage* to virtual memory systems executing on a set of client machines. In a remote memory system, the client virtual memory systems share the large memory resource. In addition, the high-speed, random access remote memory combined with high-speed networks and processors create the possibility of remote memory backing storage with performance competitive with, or better than, conventional disk backing storage, which is plagued by slow seek times.

Remote memory backing storage differs from conventional magnetic disk backing storage in several ways, and many of the differences affect the design of the client operating system. The most obvious difference between remote memory backing storage and disk backing storage involves reliability. When an operating system issues a read or write operation to a disk, the disk hardware executes the desired operation and then reports the status of the operation (e.g., success or failure). Because network hardware does not report the success or failure of a network transmission, the hardware may lose remote paging requests without signaling an error. In addition, because clients access remote memory across a network, the server may asynchronously accept requests from multiple clients. Disks, however, can only process one read/write request at a time. The operating system cannot issue another disk operation until the previous operation completes. Clients, on the other hand, may interrupt the server with a second or third request before the server finishes processing the first request. If the system does not monitor the number of outstanding requests, a client could swamp the server. Moreover, the network may deliver requests to the server out of order.

Operating systems that use magnetic disks for backing storage maintain data structures that manage the allocation and deallocation of the disk space. However, in the remote memory model, the memory server manages the backing store, which frees the client operating systems from maintaining data structures that manage the backing store.

Because disks have relatively slow seek times (i.e., random access performance), conventional virtual memory systems attempt to improve performance using techniques such as *read-ahead* or *delayed writes* to reduce the number of seek operations. Remote memory backing storage does not suffer from this problem because memory, unlike a disk, provides constant-time random access to data.

1.4 Organization of the Thesis

Chapter 2 defines terminology used throughout the remainder of the thesis. Chapter 3 presents the *Remote Memory Model* and briefly describes the three basic components of the model: the client virtual memory operating system, the communication protocol, and the memory server. Chapter 3 concludes by discussing advantages and disadvantages of systems based on the model.

Chapter 4 describes a hierarchical virtual memory operating system designed to support remote memory backing storage. Chapter 5 presents a high-speed communication protocol used by the client virtual memory system to transfer data to and from the memory server. Chapter 6 describes the design of a highly efficient memory server capable of supporting multiple client machines simultaneously. Chapter 6 also describes the high-level abstraction used by the memory server to support heterogeneous client machines.

Chapter 7 describes a prototype implementation and presents experimental results obtained from the prototype. The results demonstrate the viability of the remote memory model and show that performance is competitive with existing distributed systems.

Finally, chapter 8 presents conclusions drawn from the research and outlines directions for future research.

2. TERMINOLOGY

This chapter defines terminology used throughout the remainder of the thesis. We assume the reader has a basic understanding of operating system and network terminology. A basic description of operating system terminology can be found in Peterson *et. al.* [PS85]. Denning presents an overview of virtual memory in [Den70]. Tannenbaum [Tan81] and Comer [Com88] describe network communication and distributed systems. In addition to basic terminology, this chapter clarifies ambiguous terms by defining them in the context of this thesis.

2.1 Virtual Memory Operating Systems

An *operating system* manages a computer's resources and provides support for multiprogramming. A *virtual memory operating system* supports an abstraction called *virtual memory* where each application executes in a logical memory space much larger than the size of the machine's physical memory space. Application programs use logical *virtual addresses*, as opposed to *physical addresses*, to access memory locations. Virtual memory hardware translates virtual addresses to physical addresses to locate the data specified by a virtual address. The hardware component that provides support for virtual memory is called the *memory management unit* (MMU). An MMU typically uses an address translation table called a *segment table* or *page table* to translate virtual addresses to physical addresses (see Section 1.1.1). When the MMU encounters an invalid page table entry, the MMU raises a *page fault* exception to indicate that the virtual address could not be translated or a protection violation has occurred (e.g., a write access to a read-only page).

Virtual memory operating systems categorize data storage media as *primary storage* or *backing storage*. Primary storage refers to the memory physically located in the computer, while backing storage refers to data storage media other than primary storage. Typically the backing store is characterized by slower access times than primary storage. For most conventional systems, backing storage refers to a magnetic disk drive. The virtual memory system uses the primary storage to cache data that will be accessed in the near future and stores all other data on the backing store.

The virtual memory system divides each virtual address space into *segments* or *pages*.[1] In the case of a paged system, the virtual memory system divides the physical memory (primary storage) into fixed size *frames*. The size of a frame is determined by the hardware, while the size of a page is set by the operating system. Consequently, the size of a page may differ from the size of a frame. However, most virtual memory operating systems define the size of a page to be the same as the size of a frame.

Conventional virtual memory operating systems use the abstraction of a *process* to provide multiprogramming [PS85]. A process consists of a *virtual address space* and a *point of execution* within the address space. The virtual memory system allows multiple processes to execute simultaneously. More recent systems support *multi-threaded processes* [Ras86, OCD$^+$87, CG91]. A *thread* is defined to be a point of execution within a process' address space. All threads execute in parallel but execute at separate points in the code. The advantage of multi-threaded processes is the ability to concurrently manipulate shared data within a process.

2.2 Network Communication

Computers uses a *communication channel* to communicate with other computers. A communication channel can be a physical network (e.g., an Ethernet) or a logical (virtual) network consisting of multiple physical networks connected by gateways [Com88, Tan81].

[1]Section 1.1.1 describes segments and pages.

A *datagram* is the basic unit of transfer across a communication channel. Many conventional communication channels provide *unreliable datagram service*. Unreliable datagram service refers to a connectionless message-based data transmission model in which the communication channel allows computers to send and receive datagrams but does not guarantee that the datagrams will arrive at the destination. Instead, the communication channel only guarantees to make a *best effort* to deliver each datagram [Nar88].

Each datagram has a fixed, architecture-dependent, maximum size. The maximum size of a datagram is called the *maximum transmission unit* (MTU). Many virtual network architectures provide data transfer across multiple physical network architectures connected by gateways. Although the virtual network defines a virtual MTU for the size of a datagram, each underlying physical network architecture defines its own MTU, which may be less than the MTU specified by the virtual network architecture. In this case, the virtual network *fragments* virtual datagrams into physical datagrams, or *packets*, such that the size of each packet is less than the smallest MTU of the underlying physical network architectures.

A *protocol* defines the language that hosts use to communicate. In order for two or more hosts to communicate, they must agree upon a protocol before communication can begin. A protocol specifies the format and meaning of the information as the information travels from one host to another. Protocols also define the set of operations to perform when errors occur.

A *network architecture* is defined by a set of layered protocols called a *protocol stack* [McF76, Com88, Sta91]. Figure 2.1 illustrates the protocol stack used in the International Standards Organization Open Systems Interconnection (ISO OSI) Reference Model. When a message travels from one host to another, the message traverses the layers of the protocol stack. Each layer of the protocol stack provides a set of services and may only use the services provided by the layers below. Organizing the network communication task into multiple layers simplifies the design process by breaking the communication task into smaller, managable pieces.

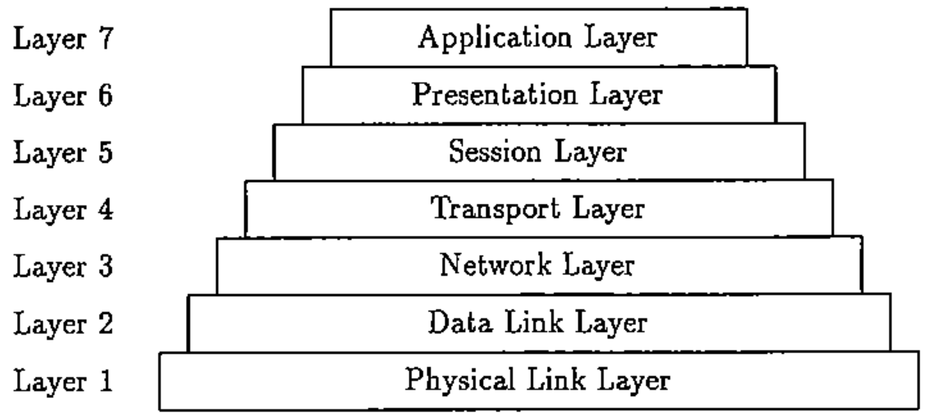

Figure 2.1 The OSI Reference Model.

Communication channels transmit data at a rate specified in *bits per second* (bps). For the purposes of this thesis, we use the relative terms *high-speed* and *high-bandwidth* to describe communication channels capable of transmitting data at rates in excess of 10 megabits per second (Mbps).[2]

Host machines (and gateways) often enqueue incoming messages until the host has a chance to process the messages. When a packet travels from one host to another it may experience one or more *queueing delays*. The time a message spends in the queue waiting to be processed is called the *queueing delay*. In the case of a virtual network where a message traverses several gateways, a message experiences a queueing delay at each gateway between the sender and the receiver.

One measure of the performance of a communication channel is the average *transmission delay*. The transmission delay is the amount of time required to transmit a message from one host to another. Another measure of performance is the *round trip time* (RTT). In the client-server model, the round trip time refers to the elapsed time between the time at which a client first sends a request message to a server and the time at which the client receives the acknowledgement or reply message from the server. We will often refer to the round trip time as the *round trip delay*.

[2]As we will see in chapter 7, a 10 Mbps communication channel provides acceptable performance (i.e., competitive with conventional systems).

2.3 Distributed Systems

A *distributed system* consists of multiple computers called *hosts*, or *nodes*, connected by a *communication channel.* A host or node is a general purpose computer that typically executes an operating system and supports user-level applications. The communication channel allows hosts to communicate information by sending and receiving messages. A *heterogeneous distributed system* consists of host machines with a variety of hardware architectures.

In a distributed system, each node provides a subset of the services available to the users of the system. Together the nodes comprise a system with full functionality and a complete set of services. In contrast, a *stand-alone* system consists of a single computer which provides a complete set of services without the aid of any other computers.

Many conventional distributed systems use the *client-server model* of computing [Com87, Tan81]. Systems based on the client-server model identify common operations or functions and define them as a *service.* A *server process* performs one or more services and allows other hosts in the distributed system to use the services it provides. A server presents a well-defined interface to the service(s) it performs. Applications throughout the distributed system know and use the well-defined interface to access the service. Application programs access the service by sending a request message across the communication channel to the server process. When an application issues a request message, the application becomes a *client* of the server. The server receives the request message from the client, performs the service requested in the message, and returns the results to the client in a reply message. For example, one node might provide a file service while a second node provides a computational service and a third node interacts with the user via a user-interface. Although none of the nodes constitute a complete system, when combined together, the nodes provide all the services available to the user.

The advantage of the client-server model is that it functions correctly when the server and the client are on the same machine (e.g., a stand-alone system) or when the server and the client are on separate machines (e.g., a distributed system). In addition, the model partitions the system functions into independent, self-contained services that form an elegant, modular system.

3. THE REMOTE MEMORY MODEL

Virtual memory operating systems afford programmers the luxury of developing applications that assume an arbitrarily large memory space. Virtual memory frees programmers from architecture-dependent memory constraints and allows them to produce highly portable, architecture-independent applications. The operating system creates the illusion of an arbitrarily large memory by mapping a small portion of the virtual memory space to physical storage. The physical storage space consists of primary memory and a backing store (typically, a magnetic disk). The operating system maps certain regions of the virtual address space to physical memory and maps the remaining regions to the backing store. The virtual memory system manages (i.e., allocates and reclaims) the primary memory and the backing store. Typically the virtual memory system maps regions that have a high probablility of being accessed in the near future to physical memory in an attempt to minimize access to the backing store.

Conventional distributed systems execute a virtual memory operating system on each node in the system. Each virtual memory system independently manages its local, private memory. No mechanism exists for the virtual memory system on one machine to use unused memory on another machine. The virtual memory systems do not interact and have no means of sharing the physical memory located on the various nodes. That is, conventional distributed systems preallocate and assign private physical memory to each machine instead of allowing all the machines to share the physical memory resources. Consequently, the virtual memory system only allows applications to use local memory. When the applications exceed the capacity of the local memory, the virtual memory system moves application text and data to the machine's private backing store instead of using unused memory elsewhere in the system.

Current trends in memory technology create the possibility of machines with large memories. We envision a distributed system with enough physical memory to back much, if not all, of the virtual memory being used by applications throughout a distributed system. This chapter describes a new virtual memory model called the *remote memory model*. The remote memory model defines a distributed system in which all the machines share a large, physical memory resource. The system provides simple and efficient mechanisms for accessing the large memory resource and allows every machine's virtual memory system to access the memory resource. Instead of preallocating and installing all the physical memory resources as private memory on the individual machines, the model combines some of the physical memory resources together into a single, large, globally accessible, memory resource and provides efficient access to the memory. Consequently, the virtual memory systems all share the memory resource.

The remainder of this chapter describes the remote memory model and the assumptions we make about the model. It identifies the properties of the model and describes advantages and disadvantages of building systems based on the model.

3.1 The Model

The *remote memory model* describes a new architecture for designing distributed systems. The model proposes a new virtual memory architecture that enhances both the functionality and the performance of distributed systems. It defines the various components of the system, the role each component plays, and the interaction between the various components.

Figure 3.1 illustrates an example remote memory model architecture. The system consists of multiple *client* machines, one or more *memory server* machines, various other servers (e.g., time servers, name servers, or file servers), and a communication channel interconnecting all the machines. In the remote memory model, memory

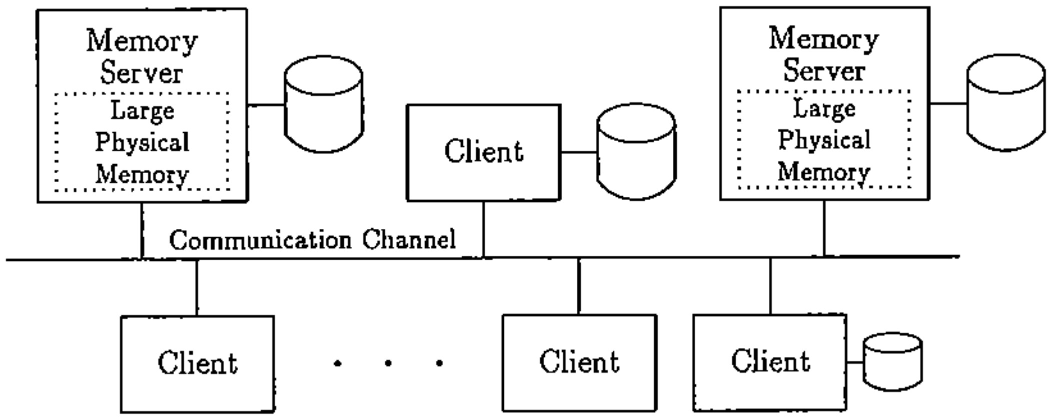

Figure 3.1 An example remote memory model architecture.

server machines provide additional memory storage space for the virtual memory systems executing on the client machines. Instead of parceling out the memory resources to individual machines where the memory is installed as private memory, the remote memory model provides a large memory resource which all the clients share. In the remote memory model, client machines access the memory server to obtain additional storage space.

Each client machine has a local memory capable of satisfying the client's normal processing demands. However, for jobs requiring large amounts of memory, clients use the large memory of the memory server for additional storage space. When the memory requirements of an application exceed the capacity of the local memory, the client's virtual memory system stores some of the application's data on the memory server. When the application attempts to access data not cached in the local memory, the virtual memory system intercepts the access, transfers the desired data from the memory server to the client's local memory, and resumes execution. To free local memory, the virtual memory system periodically transfers data from the local memory to the memory server. Each client's local memory functions as a high-speed cache of the large, shared, memory resource on the memory server.

All the client machines connect to the communication channel and contain the necessary hardware support required to send and receive messages across the communication channel. A client machine may be anything from a small microprocessor (e.g., a sensor device) to a multi-user time-sharing system (e.g., a workstation or mainframe). Each client machine executes a virtual memory system, typically assisted by address translation hardware. The virtual memory system has the functionality required to access the large storage space located on the memory server. That is, the virtual memory system understands the protocol used to communicate with the server as well as the operations provided by the server. The virtual memory system hides the location of the data (i.e., whether the data exists in local memory or remote memory) from the application. The client's virtual memory system, with the help of the memory server, creates the large virtual address spaces visible to the applications.

Large memory machines called *memory servers* provide shared, high-speed remote memory storage to the virtual memory systems executing on the client machines. Memory servers do not execute user applications. Instead, memory servers are dedicated machines whose sole purpose is to provide remote data storage. Each memory server machine has a large amount of storage space. The storage space consists of physical memory and a mass storage device (e.g., disk drive). The total size of the storage space on a memory server must exceed the expected maximum combined memory usage of all the client machines. Clearly, the maximum memory usage depends on the specific system configuration.

Although memory servers may use secondary storage to increase the total amount of storage space, memory servers do not require virtual memory support from the hardware. For performance or economic reasons the designer of a memory server machine may chose to omit the MMU. In this case, the memory server must provide all virtual memory management (e.g., swapping to secondary storage) in software. However, each memory server machine must have the network communication hardware required to communicate with the client machines it serves. Because memory server machines only require the aforementioned hardware, they are relatively simple machines (i.e., they do not require serial ports, video boards, monitors, or an MMU) and can be viewed as a "black box" providing high-speed data storage.

The communication channel provides every client with the ability to send and receive messages to and from the memory server. The communication channel acts like a high-speed bus or backplane connecting multiple processors to a shared memory. However, the communication channel differs from a bus in several ways. The model assumes the communication channel provides unreliable datagram delivery. That is, the model only assumes the communication channel makes a *best effort* to deliver each datagram [Nar88].

Assuming unreliable datagram service allows the model to encompass a wide variety of network architectures. For example, the unreliability assumption allows the model to use both reliable and unreliable network hardware. Similarly, the datagram delivery assumption allows the model to use network architectures that provide datagram delivery (packets with a fixed maximum size), message delivery (logical messages of arbitrary size), or byte-stream delivery (using record marking to emulate datagrams). Moreover, the model does not restrict the communication channel to a low-level protocol in the protocol stack. Consequently, the model may use a link-level physical network architecture (e.g., Ethernet) or a transport-level virtual network architecture (e.g., TCP/IP) as the communication channel.

Although the example communication channel shown in Figure 3.1 appears to provide broadcast capabilities, the remote memory model does not assume a broadcast facility. The model only assumes that a communication path exists from every client to the memory server. Consequently, other communication technologies, such as point-to-point network architectures, may serve as the communication channel.

The remote memory model provides an alternative to conventional distributed systems. The model approaches virtual memory differently than conventional distributed systems by allowing clients to share a large, globally accessible, memory resource. Memory servers act as monitors for the memory resource, allocating memory to clients based on their needs. The remote memory model also provides the client virtual memory systems with the opportunity to interact via shared data. The above functionality differences together with high performance make the remote memory model an attractive alternative to conventional distributed systems.

3.1.1 Design Decisions

The general form of the model, described in the previous section, encompasses a wide variety of system architectures. This section further defines various unspecified aspects of the model. In particular, we make some design decisions that result in a more precise or focused definition of the model. However, the design decisions are not part of the model. They simply refine the definition of the model to provide the functionality and performance we desire.

The continued proliferation of computer architectures make heterogeneous systems a reality that cannot be ignored. Consequently, the model should support heterogeneous distributed systems. The size of a virtual address space, physical memory size, page size, byte order, and word size may differ from machine to machine. Moreover, client machines may execute heterogeneous operating systems, each with its own virtual address space configuration, process structure, and replacement policy. Despite these differences, memory servers must provide remote memory to all client machines, regardless of the client's architecture or operating system.

We assume the majority of clients execute a multi-user or multi-tasking virtual memory operating system with support for multiple virtual address spaces. Although clients use remote memory instead of a local disk for backing storage, the model does not assume diskless clients. Client machines may use a local disk for permanent storage (e.g., a local file system). To simplify the model and the design of the client virtual memory system, we assume each client only accesses one memory server even if multiple servers exist.[1]

[1]The single server restriction also facilitates rapid prototyping and experimental evaluation of systems based on the model.

Each memory server machine has a large physical memory capable of backing most, if not all, of the virtual memory being used by the applications throughout the system. Recent advances in memory technology allow each memory server to have hundreds of megabytes, gigabytes, or possibly even terabytes of physical memory. As we mentioned earlier, each memory server must have a storage space whose total size exceeds the maximum memory usage of the clients. Consequently, a memory server may use secondary storage to increase the total storage space on the server. However, for performance reasons, the memory server should have a large physical memory to reduce the number of accesses to secondary storage.

One of the goals of the remote memory model is to provide high-speed access to additional memory storage. Consequently, the communication channel should have a high bandwidth and low delay. Although the communication channel may be unreliable, the channel should exhibit a relatively low error rate. These requirements typically restrict the communication channel to high-speed local area networks (LANs) or metropolitan area networks (MANs). Example communication channels include Ethernet (10 Mbps), proNET (10 or 80 Mbps), and FDDI (100 Mbps) [Dig80, RHF90]. Virtual network architectures (e.g., TCP/IP networks, OSI networks) may be used as the communication channel as long as they adhere to the high-speed communication requirement.

3.1.2 Characteristics of the Model

Having described the remote memory model, this section focuses on characteristics of the model and briefly examines advantages and disadvantages of designing systems based on the model.

Perhaps the most apparent difference between the remote memory model and conventional distributed systems is the high-speed data storage offered by the memory server. Client applications that require large amounts of memory obtain additional high-speed memory from the memory server. In addition to providing a large, high-speed storage space, the remote memory model has the following desirable characteristics:

Shared Access

Each memory server provides data storage to multiple client machines simultaneously. All the client virtual memory systems concurrently access the memory server to store and retrieve data. Instead of preallocating a fixed amount of memory to each client, the memory server allows the client virtual memory systems to share the memory resource by dynamically assigning and reassigning the memory resource to clients based on their changing needs. In addition, the server does not limit the number of clients it will serve. Each memory server allows an arbitrary number of clients to access its memory.

Shared Data

The remote memory server provides a centralized memory, accessible to all client machines. The globally accessible memory resource provides the opportunity for clients to efficiently share data. Although the model permits data sharing among clients, the model does not specify the mechanisms used to share data. Depending on the type of data sharing desired, the memory server may implement data sharing mechanisms that allow read-only sharing, read-write sharing, or no sharing [Li86, LH89]. Because multiple client machines often execute a given application, use a given library, or memory map a given file (e.g., a font file or a static database), even a simple read-only sharing mechanism significantly reduces the amount of server memory consumed by client data. A more complex sharing mechanism might use the server as a centralized monitor to implement read-write distributed shared memory. The centralized monitor would grant or deny access to shared regions and provide the coherency control required to maintain consistent copies of the data on all the client machines.

In short, the model allows the memory server to support several shared data models.

Arbitrarily Large Storage Capacity

Memory servers employ virtual memory techniques to create an arbitrarily large memory resource in which clients may store data. Although memory servers have very large memories, the amount of memory on any given server remains fixed and depends on the server's underlying architecture and hardware configuration. To provide an arbitrarily large storage space, independent from the physical memory size of the memory server, a memory server uses one or more secondary storage devices (e.g., disk drives) and a replacement policy to substantially enlarge the server's storage capacity. When the client machines collectively exhaust the physical memory on the server, the server transfers client data to secondary storage, thereby freeing physical memory for additional client data. As a result, client machines are completely unaware of the two-level storage space.

Memory servers hide the size of their physical memory from the clients they serve much like a virtual memory operating system hides the size of the underlying machine's physical memory from the applications executing on the machine. In a virtual memory operating system the machine's physical memory size does not constrain the amount of virtual memory an application can obtain. Similarly, the memory server's physical memory size does not restrict the amount of backing storage a client can obtain. Consequently, a client may use any memory server, regardless of the server's physical memory size.

Offloading File Server

The remote memory model improves file system performance by removing paging activity from the file system. Similarly, memory servers improve virtual memory system performance by providing high-speed backing storage. Removing paging activity from the file system significantly reduces contention for the disk and eliminates many extra head movement operations. Virtual memory systems that use a file system for backing storage compete with all the user-level processes for the file system's resource. In the case of a remote file server, the virtual memory system competes with the user-level processes of all the machines in the system. The file system gives no special privileges or priority to requests from the virtual memory system because it cannot distinguish between user-level file access and virtual memory system paging operations.

File systems often optimize performance for the most common file access patterns. Many file systems attempt to improve performance with techniques such as *read-ahead*. Read-ahead assumes sequential file access and prefetches extra data each time an application reads from a file. However, in many cases, the paging activity generated by the virtual memory system results in random data access as opposed to sequential data access. When applied to paging activity, prefetching wastes valuable buffer space and degrades both paging and file system performance. Separating paging activity from file activity allows us to implement each operation efficiently. Unlike file servers, memory servers understand paging activity and can be designed to make intelligent decisions regarding storage and retrieval of data.

Multiple Forms of Remote Memory

The remote memory model does not define the operations allowed on remote memory, nor does it specify the behavior or reliability of remote memory in all situations. Consequently, the remote memory model allows system designers to define the operations and reliability provided by remote memory to meet the needs of their particular system. We already mentioned several possible semantics for providing shared data between clients. The model also allows the memory server to store and retrieve data with various reliability guarantees. For example, the client virtual memory systems may view remote memory as a less efficient form of local memory, but similar in all other respects. Given this definition, the memory server provides high-speed volatile storage similar to the client's high-speed volatile local memory. A different definition of remote memory may require *reliable storage.* In this case, the memory server must use a non-volatile storage device to store a copy of all the data written to remote memory. Another definition may require *reliable storage* and *reliable retrieval.* In this case, multiple memory servers may cooperate to provide fault tolerance and insure that clients can access remote memory at all times.

Exploits Technological Advances

Network bandwidth, CPU speeds, and computer memory sizes are increasing at a rapid rate. The remote memory model exploits these particular hardware technologies and will continue to exhibit better performance as the technology advances. The average seek time on a magnetic disk, however, has remained relatively constant. In the future, we expect the remote memory model to offer performance several times faster than systems that page to a local disk.

These advantages make the remote memory model an attractive model for designing distributed systems. However, the model is not without its drawbacks. Because memory servers provide a centralized memory service, they present a potential bottleneck. If the number of requests becomes too large for the server to handle, the server's ability to respond quickly and efficiently to requests will deteriorate and degrade the performance of the client virtual memory systems. Because all clients access remote memory across the communication channel, the communication channel poses a potential bottleneck. As the number of clients increase, the amount of data traveling to and from the server increases and consumes a significant portion of the communication channel's bandwidth. Moreover, remote memory traffic may compete with all other network traffic for the communication channel's bandwidth. Despite these potential problems, the remote memory model's attractive properties combined with the possibility of performance competitive with, or better than, conventional magnetic disk backing storage make the remote memory model an attractive model for designing distributed systems.

3.2 Summary

This chapter describes a new model for designing distributed systems. The model proposes a new virtual memory architecture in which dedicated, large-memory machines serve as backing store for virtual memory systems operating on a set of heterogeneous client machines. The dedicated memory server allows sharing of the large physical memory resource and provides fast access to data.

The remote memory model has several desirable properties. Memory servers provide high-speed data storage to virtual memory systems executing on heterogeneous client machines. The centralized nature of the memory server provides clients with the opportunity to share data. The remote memory model improves file system performance by offloading the file server. Separating file activity from paging activity allows us to implement each operation efficiently. Finally, the model appears promising for the future because it exploits the rapid technology advances in network bandwidth, CPU speeds, and memory size. In short, the remote memory model provides an attractive alternative for designing distributed systems.

4. A CLIENT VIRTUAL MEMORY SYSTEM

In the remote memory model, the client virtual memory system communicates with the memory server to access remote memory backing storage. As you will recall from the description of the remote memory model presented in chapter 3, each memory server provides remote memory backing storage to heterogeneous client machines executing a variety of virtual memory operating systems. Consequently, the only requirement of the client operating system is that it must provide support for remote memory backing storage. The virtual memory system must understand (and use) the set of operations provided by the memory server. In addition, the operating system must implement the communication protocol used to communicate with the memory server.

This chapter presents the design of a virtual memory operating system with support for remote memory backing storage. The system presented here serves as an example of how to incorporate remote memory backing storage into a virtual memory operating system. The same mechanisms and approach could be applied to a wide variety of existing operating systems, thereby allowing them to make use of remote memory backing storage.

4.1 Design Goals

In the remote memory model, the virtual memory component of the client operating system provides the support required to access remote memory backing storage. While designing the client virtual memory system, we identified several additional design goals indirectly related to the basic goal of incorporating remote memory backing storage into the virtual memory system. In particular, we envisioned a client virtual memory operating system with the following characteristics:

Architecture Independence

In order to execute the operating system on a wide variety of client architectures, the operating system must exhibit a high degree of portability. In particular, the virtual memory system must not depend on the underlying hardware architecture. Clearly, architecture dependencies will exist, but the design should limit and isolate them as much as possible.

Hierarchical Design

Operating systems are large, complicated pieces of software. Experience with the simplicity, flexibility, and clarity of existing hierarchically-designed operating systems demonstrates the benefits of designing operating systems in a hierarchical fashion [Com84, Dij68, Lis72, SAG+72].

Multiple Threads of Control

The system should support multi-threaded user applications and allow multiple threads of control within the kernel. Multiple threads of control within a user application allow concurrent manipulation of shared data within the application, and multiple threads of control within the kernel allow the kernel to execute several tasks concurrently. Multiple threads of control allow cooperating tasks to communicate efficiently via shared memory. In addition, many operations can be implemented simply and elegantly when viewed as concurrent threads of control [Com87, Tan87].

An Efficient Remote Paging Mechanism

The performance of the virtual memory system depends on the communication between the client and the remote memory server. To maximize performance, the paging mechanism must be efficient. In addition, the virtual memory system should exhibit flexibility and permit system designers to substitute alternate forms of backing storage.

Sections 4.2 – 4.4 examine each of these design goals and their relationship to the remote memory model. Section 4.2 shows how the operating system uses a hierarchical design to separate the various functions performed by the system into distinct components. In particular, the hierarchical design allows the system to separate virtual memory management from the backing store I/O mechanism. Separating virtual memory management from backing store I/O allows the system to use new forms of backing storage, including remote memory. Moreover, the modularity that results from a hierarchical design permits replacement of various system components such as the page replacement mechanism. The ability to change the page replacement mechanism allows us to experiment with new replacement policies and examine the interaction between the replacement policy and the backing store; in this case, remote memory backing store. Section 4.3 outlines the design of an architecture interface layer that provides architecture independence and reduces the effort required to port the operating system to the heterogeneous client architectures. Section 4.4 shows how the kernel uses multiple threads of control executing in a shared address space to efficiently send and receive data to and from the memory server across the communication channel. The threads share data structures and use the kernel's interprocess communication and synchronization primitives to communicate with each other. Together the threads implement the protocol used to communicate with the memory server. Finally, section 4.5 describes the design of the virtual memory component of the *VM Xinu operating system* and shows how it achieves the design goals.

Although this chapter describes the design of a client operating system with support for remote memory backing storage, many of the design goals presented above could be applied to the design of a conventional virtual memory operating system in which magnetic disks, as opposed to remote memory, serve as the backing store.

4.2 Hierarchical Design

A hierarchically-designed operating system partitions operating system functions into distinct components and organizes the components into a layered hierarchy. Each layer of the hierarchy builds on the functionality provided by the lower layers. A hierarchical design, like a modular design, identifies functions that perform closely related operations and combines them into layers with well-defined interfaces and semantics. However, unlike modular designs, hierarchical designs specify the dependencies between layers by defining a hierarchical relationship between the layers. The core layer of a hierarchically-designed operating system consists of the small set of primitive operations supported by the hardware. Each layer builds on previous layers to provide additional functionality.

Because operating systems are large, complex pieces of software, the components of the system often interact in complicated and unexpected ways. Hierarchically-designed operating systems clearly define the interaction between components, making the system easier to understand. Hierarchically-designed operating systems collect the functions that define each policy (scheduling policy, memory allocation policy, page replacement policy, etc.) together into a layer, allowing the designer to easily identify the functions that implement a given set of operations or define a given policy. In particular, the hierarchical layering allows the virtual memory system to be modified to support new types of backing store without fear of breaking other system components. The dependency information provided by the hierarchy clearly identifies the components that may be affected by the change.

As the name implies, a hierarchical design only refers to the design of the system, not the flow of control. Unlike network layering, a hierarchical design does not specify the procedures a particular layer may call at run time. Layered network models usually impose strict layering on the run-time execution, requiring the thread of execution to sequentially traverse each layer in the network model. A hierarchical design does not impose such stringent restrictions on the run-time execution.

4.2.1 Incorporating Virtual Memory into a Hierarchical Design

Building on the experience obtained from hierarchically-designed operating systems, we designed a virtual memory operating system that incorporates virtual memory into a hierarchical design. We call the system *VM Xinu* because it has its origins in the Xinu operating system [Com84, Com87]. The hierarchical design found in the Xinu operating system is shown in Figure 4.1. All Xinu processes share a single address space and execute code directly out of the kernel text region. Consequently, the memory management layer only manages a single physical address space.

Figure 4.2 illustrates VM Xinu's hierarchical design and shows the dependencies between the virtual memory system components and the other components of the system. The core of the hierarchy consists of machine-specific hardware and an architecture interface layer. The architecture interface layer hides the underlying virtual memory hardware from the upper layers. It presents a well-defined virtual memory interface to all the upper-level memory management routines, thereby allowing them to remain independent from the underlying hardware.

The following sections describe the various layers of the hierarchy. The description begins with the process management layer and delays the description of the hardware layer and architecture interface layer until section 4.3 where the topic of architecture independence is discussed.

4.2.2 Process Management Layers

The major differences between the non-virtual memory hierarchy and the virtual memory hierarchy occur in the memory management and process management layers. Although it seems obvious that introducing virtual memory into a hierarchal design requires modifications to the memory management layers, it may not be obvious how the introduction of virtual memory affects the process management layers.

In the system without virtual memory (Figure 4.1), memory management resides at the core of the hierarchy and does not depend on process management. The process management layer resides above the memory management layer because it depends

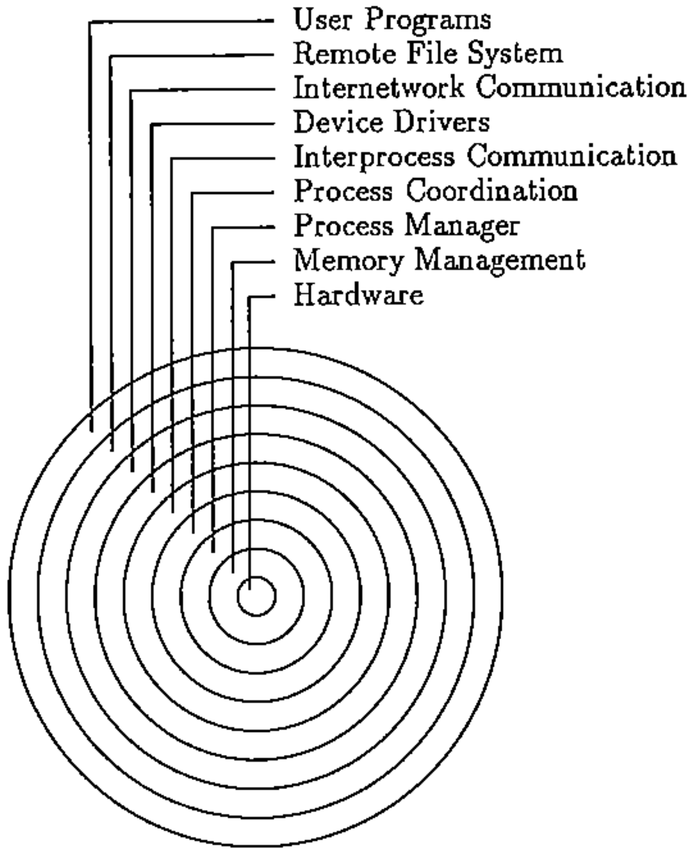

Figure 4.1 The hierarchy of layers in the Xinu operating system.

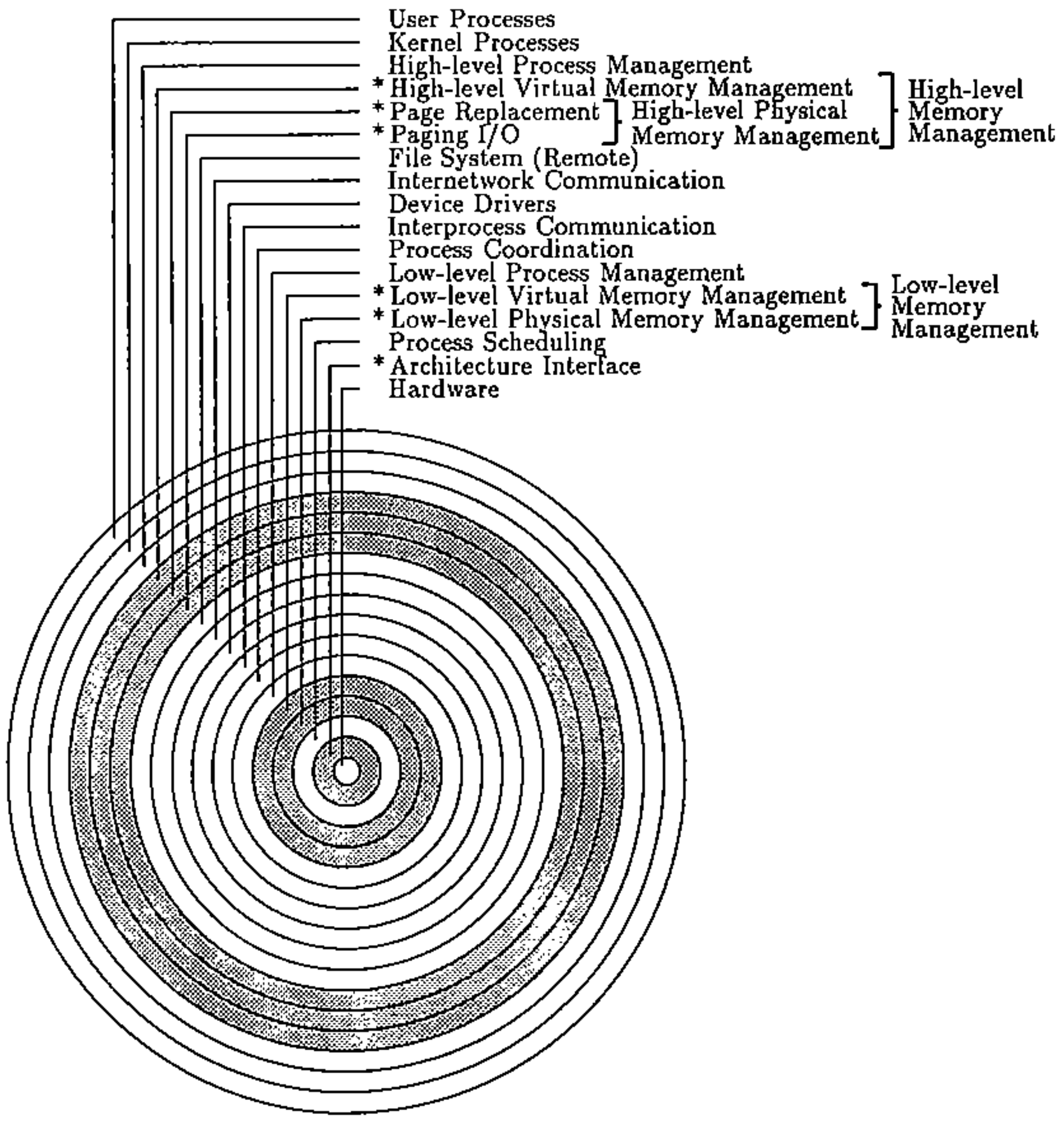
User Processes
Kernel Processes
High-level Process Management
*High-level Virtual Memory Management
*Page Replacement
*Paging I/O
High-level Physical Memory Management
High-level Memory Management
File System (Remote)
Internetwork Communication
Device Drivers
Interprocess Communication
Process Coordination
Low-level Process Management
*Low-level Virtual Memory Management
*Low-level Physical Memory Management
Low-level Memory Management
Process Scheduling
*Architecture Interface
Hardware

Figure 4.2 A layering model for a hierarchically-designed virtual memory operating system. The *'s identify the virtual memory system components of the hierarchy.

on the memory management layer to allocate stack space for newly created processes. However, in a virtual memory system, the memory management system requires functionality found in the process management layer to suspend processes that request memory when no memory is available. That is, memory management depends on functionality provided by process management, creating a circular dependency. The hierarchical design shown in Figure 4.2 resolves the circular dependency by breaking the process management layer into three components: *process scheduling, low-level process management*, and *high-level process management.*

The *process scheduling* component of the system moves processes between process states and schedules the ready processes for execution based on a *scheduling policy.* The design places the process scheduling layer at the core of the hierarchy because the memory management layer depends on the functionality that it provides. To understand the dependency, imagine an application consumes all the available physical memory. The virtual memory system must then free memory by writing data to the backing store. The memory management routines choose a page for replacement and issue a write operation to store the page on the backing store. After the memory management routines issue the write, they must wait for the secondary storage device to return the status of the write operation. To avoid busy-waiting while waiting for the return status, the memory management routines invoke the process scheduling layer to suspend execution of the waiting process and resume execution of the next ready process. In a non-virtual memory system, the memory management layer signals an error when it exhausts the physical memory. However, in a virtual memory system the memory management layer must suspend the process until memory becomes available.

The *low-level process management routines* combined with the *high-level process management routines* implement process creation and process termination. The low-level routines supply basic process creation operations that initialize a virtual address space, allocate stack space, and fill in the process table. The low-level routines also provide termination operations that unmap virtual address spaces, release stack space, and free process table entries. The high-level routines provide the ability to dynamically load user code from the file system and execute it. Consequently, the high-level process management layer resides above the file system layer in the hierarchy. The high-level process management layer also serves as the interface between user level applications and the low-level process creation/termination primitives.

4.2.3 Memory Management Layers

In a virtual memory system, the memory management component must handle the allocation of both physical and virtual memory. The design divides the memory management component into two layers: *low-level memory management* and *high-level memory management* (see Figure 4.2). Each of these two layers is subdivided into a *physical memory management sublayer* and a *virtual memory management sublayer.* The design separates low-level details, such as mapping physical frames to virtual pages, from high-level abstractions, such as the layout of a virtual address space. In particular, the separation allows the paging component of the virtual memory system to reside above the I/O system and the file system. As a result, the paging component may use a wide range of backing storage media including disk drives, files, or remote memory.

The *low-level physical memory management* routines handle the allocation and reclamation of physical frames. When the system needs a physical frame to back a virtual page, it calls a low-level physical memory management routine to obtain an available frame. If a process needs a physical page and no pages are available, then the low-level physical memory management layer suspends execution of the faulting process by invoking functions at the scheduling level. The low-level physical memory management routines never perform any paging operations (i.e., move data to or from backing storage). The low-level physical memory management routines also allocate and free kernel memory.

The *low-level virtual memory management* routines provide the support required by the process creation layer to initialize a new virtual address space. In particular, the low-level virtual memory management routines create and initialize the stack region of a new virtual address space. The routines use the functionality provided by the low-level physical memory management layer to allocate the physical frames assigned to the stack region.

The *high-level physical memory management* layer performs all the operations related to paging. Figure 4.2 depicts high-level physical memory management as two distinct sublayers: the *paging layer* and the *page replacement layer*. The *paging layer* transfers pages to and from the backing store. Because the virtual memory system performs paging at a high-level in the hierarchy, it can use the file system layer, network communication layer, or device driver layer to access backing storage. This ability allows the system to page to a wide variety of secondary storage mediums.

The *page replacement layer* decides which virtual pages should remain in the local physical memory and writes all other virtual pages to the backing store. The page replacement layer uses the replacement policy to choose virtual pages to replace and invokes the paging layer to transfer the data from physical memory to the backing store. When the system attempts to access a virtual page residing on the backing store, the page replacement layer calls the low-level physical memory management layer to obtain an available physical page and then invokes the paging layer to retrieve the data and fill in the newly-allocated physical page.

The *high-level virtual memory management* layer defines the layout of a virtual address space by dividing the address space into special purpose segments and regulating the allocation and reclamation of memory within each segment. Example regions include a heap region, shared memory regions, and memory-mapped file regions. The high-level virtual memory management routines regulate access to these virtual regions and allow user-level applications to acquire memory from the regions.

4.3 Architecture Independence

Although a hierarchical design clarifies the relationship between the various system components, designing and implementing an operating system is still a difficult and time-consuming task. To reduce the effort required to port the system to new architectures, the design should minimize the amount of machine-dependent code. That is, the design should isolate and limit the components of the system that depend on the underlying architecture as much as possible. Many conventional operating systems use high-level abstractions to hide hardware-specific devices. such as disk drives, printers, and terminals from the rest of the operating system [Bac86, LKKQ89, Ras86, OCD$^+$87, LL82, AR89]. Our goal was to hide the underlying virtual memory support provided by the hardware from the virtual memory system.

To keep the virtual memory system independent from the hardware, the operating system defines an *abstract virtual memory machine.* The abstraction hides the underlying virtual memory hardware from the virtual memory system. Consequently, the virtual memory system requires no modifications when porting to a new hardware architecture. In addition, the small number of operations provided by the abstraction minimizes the effort required to port the abstract virtual memory machine to new architectures. Although the hardware only provides a subset of the operations defined by the abstract machine, the architecture interface layer compensates for the missing functionality by implementing the remainder of the abstract machine in software.

4.3.1 The Hardware Layer

The intersection of the functionality found in conventional hardware architectures defines the support provided by the hardware layer of the hierarchy. Using the intersection as the basis for the hardware layer increases the number of hardware platforms to which the system can be ported. Conventional architectures translate virtual addresses to physical addresses using page table entries that minimally contain the following information:

Valid Bit: indicating whether or not the entry contains a valid mapping for the hardware to use when translating a virtual address to a physical address.

Protection Bits: specifying the type of access allowed. Many architectures maintain two protection values for each virtual page; one value for kernel mode and one value for user mode. Protection values are chosen from three values that correspond to: *no access*, *read access*, and *read/write access*.

Frame Number: specifying the physical frame associated with the virtual page.

The hardware layer (i.e., the underlying architecture) must support these bits. The hardware uses the valid bit, the protection bits, and the frame number to translate virtual addresses to physical addresses. If the valid bit is not set or if the protection bits prohibit access to a page, the hardware signals an address translation error and invokes an interrupt handler to correct the problem.

Most architectures cannot map an entire virtual address space because they do not have sufficient memory to store the mapping tables. However, most architectures can map at least two disjoint regions of a virtual address space. For example, the VAX is capable of mapping three disjoint regions, while the Sun 3/50 is capable of mapping 255 disjoint regions. To allow maximum portability, the design assumes the hardware supports no more than two disjoint regions.

To improve performance, conventional architectures cache page table entries automatically in high-speed registers or memory commonly called a *translation lookaside buffer* (TLB). The abstract virtual memory machine assumes the hardware supports a TLB as well as methods for flushing the TLB. The upper layers use the flushing operation to insure that the hardware always uses current mappings rather than old cached mappings. If the hardware does not support a TLB, the flush operation maps to the null operation.

Similarly, the operating system assumes the hardware supports a *data cache* and *instruction cache*. Again, if the hardware does not support these caches, the flushing operations map to the null operation.

4.3.2 The Architecture Interface Layer

The architecture interface layer supplies the remainder of the abstract virtual memory machine's functionality and provides the higher layers of the hierarchy with a well-defined interface to the underlying architecture.

The architecture interface layer expands the information associated with each page table entry to aid the virtual memory system with tasks such as page replacement. The abstract machine's page table entries contain the following values in addition to the hardware values discussed earlier:

Modified Bit: indicating whether or not the virtual page has been modified.

Referenced Bit: indicating whether or not the virtual page has been referenced.[1]

In-Memory Bit: indicating whether or not there is a physical frame associated with the virtual page.

Initialization Bit: indicating how to initialize uninitialized pages the first time they are referenced; initialization may be *zero-on-demand* or *garbage-on-demand.*

Most of the architectures mentioned earlier provide MMU support for the modified and referenced bits, making the architecture interface layer's implementation of these bits trivial. However, architectures like the MIPs chip do not provide a referenced or modified bit. Instead, the hardware provides special instructions, registers, and interrupts designed to help the software simulate the referenced and modified bits.

Because the abstract machine defines an abstract page table entry, each entry could be defined to contain several more values. However, the values given above

[1]Because some hardware architectures do not support reference bits, the current implementation (described in chapter 7) does not require a referenced bit.

provide a sufficient set of building blocks on which one can construct a wide variety of page replacement schemes, virtual memory mapping strategies, and backing store mechanisms.

Before defining the other mechanisms supported by the architecture interface layer, a brief description of the abstract virtual memory machine's notion of a process is required. The abstract virtual memory machine supports multi-threaded processes (i.e., multiple threads of control executing in a virtual address space).[2] Portions of the address space are shared by all threads executing in the address space, while other portions of the address space are private to each thread. In particular, the abstract machine assumes a thread maintains state information on a private stack region. As stated earlier, the underlying hardware layer can map at least two disjoint regions of a virtual address space to physical memory. The abstract machine associates each region in a virtual address space with a thread (e.g., the private regions) or with the virtual address space (e.g., the regions shared by all threads). All the threads in a virtual address space share the page table entries associated with the virtual address space but have separate page table entries for their private virtual memory regions.

The architecture interface layer includes mechanisms to access the abstract machine's page table entries. Because the abstract machine understands virtual address spaces and threads, a virtual address space identifier or a thread identifier along with a virtual address uniquely identifies a page table entry. The architecture interface layer provides a routine that maps a virtual address space/thread identifier and a virtual address onto a page table entry *handle*. The high-level memory management routines use the handle to access or modify the values in an abstract machine page table entry.

The architecture interface layer also provides ways to change from one virtual address space to another with a single operation. The operating system uses the operation at context switch time to insure that the hardware has all the page table entries it needs to translate virtual addresses to physical addresses for the current

[2]Threads are discussed in more detail in section 4.4.

thread. It allows the architecture interface layer to move page table entries between the MMU and memory. For example, on the Sun 3/50 architecture these routines move the necessary page table entries into the MMU chip, possibly moving other entries from the MMU to memory. Because the page tables reside in memory on the VAX architecture, changing the page table base registers causes the hardware to use a new set of page tables.

The functionality found in the hardware layer and the architecture interface layer, together with a well-defined interface, comprise the abstract virtual memory machine. All higher level memory management routines manipulate the virtual memory hardware using the interface provided by the abstract virtual memory machine.

4.4 Process Management

Process management in the VM Xinu system differs from other conventional operating systems. The differences provide added functionality for user-level applications and allow the kernel to support remote paging cleanly and efficiently. Section 4.2.2 showed how process management fit in the virtual memory hierarchy, and section 4.3 defined the multi-threaded process support provided by the abstract virtual memory machine. This section describes how the process management layers build on the abstract machine to attain the design goals presented at the beginning of the chapter.

The process model used in many conventional operating systems ties the lifetime of a process to the threads of control within the process [RT74, LL82, GMS88]. When the threads of control within a process terminate, the system removes the process and its address space. VM Xinu uses a different process model. The process model in VM Xinu separates the computation on the data from the lifetime of the data and allows high-level procedures to tie them together when desired. In addition to providing support for the conventional process model, the system provides support for object-based models that permit inactive data objects void of any computational threads.

VM Xinu calls the data component an *address space* and the computations on the data *threads.* Given address spaces and threads, higher-level entities such as conventional processes, multi-threaded processes, active/inactive objects [DJA88], and passive objects [ABLN85, Dew88] can be constructed. For example, conventional processes, like those found in UNIX, can be formed by creating a single address space and a single thread of control within the address space. Address spaces and threads also provide the mechanisms needed to implement kernel threads. Placing kernel code and data in a distinct *kernel address space* allows multiple threads of control to execute concurrently within the kernel.

Address spaces do not depend on threads in any way. An address space may have many active threads of computation, or none at all. The data in an address space may continue to exist even after all threads of computation on the data complete. An address space may also exist long before an application starts a thread of computation on the data in the address space.

Threads, on the other hand, depend on address spaces. Each thread inhabits exactly one address space. The lifetime of a thread is limited by the lifetime of the address space. Before removing an address space, the operating system terminates all threads executing in the address space.

The next two sections describe the internal structure of address spaces and threads.

4.4.1 Address Spaces

An address space consists of two disjoint virtual memory regions. Each address space defines a mapping from virtual pages to physical frames. The architecture interface layer provides the functionality required to manipulate the address space mappings using architecture-independent routines. In addition, the virtual memory system defines a template for data placement within an address space. Figure 4.3 illustrates the arrangement of data in an address space.

The first region in an address space contains the text and data portion of a user application and begins at a fixed minimum virtual memory address. The user heap

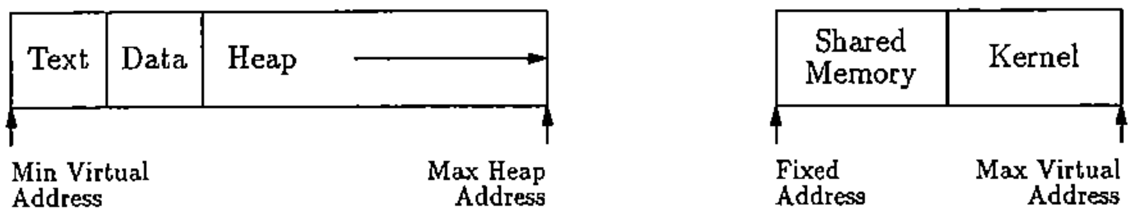

Figure 4.3 The structure of an address space in VM Xinu.

area follows the data area and extends to a maximum user heap address. The second region of the address space contains a fixed-size shared memory region used for sharing data between address spaces. The virtual memory system also maps the kernel's text, data, and heap into the second region of every virtual address space. Although the kernel appears in every address space, the protection bits on the kernel region only allow access to the kernel region while executing in kernel mode. User applications change to kernel mode by issuing a *trap* instruction to trap to a kernel routine. The virtual memory system maintains almost no state information for an address space other than the virtual to physical address mappings, allowing the virtual memory system to support many address spaces.

4.4.2 Threads

VM Xinu defines a thread as a point of execution within an address space. A thread consists of state information that describes the thread's status and the thread's private stack region. All threads executing in the same address space share the text, data, and heap region of the address space, but each has its own private stack region as shown in Figure 4.4. A thread's private stack space together with the address space in which the thread executes comprise the entire virtual address space visible to the thread. VM Xinu places each thread's private data immediately before the shared memory region in the address space. The resulting virtual address space can be easily mapped to the two disjoint virtual memory regions supported by the abstract virtual memory machine described in section 4.3.

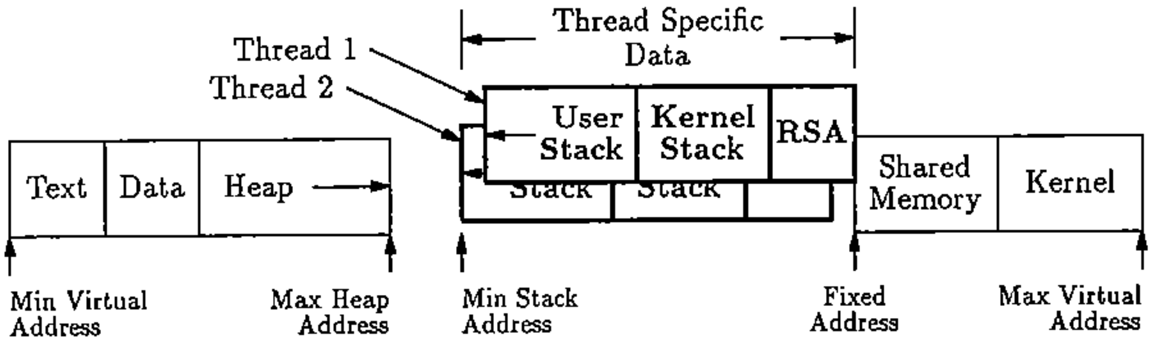

Figure 4.4 The location of thread specific data within an address space.

VM Xinu divides a thread's stack space into three sections: a user stack, a kernel stack, and a reserved shared area (RSA). A thread operates in user mode when executing user programs and in kernel mode when executing kernel calls, using the appropriate stack for each mode of operation. Both the kernel and user threads have access to the RSA area and use the area to efficiently pass data between the kernel and the user thread. The RSA area allows the user to efficiently implement I/O operations by eliminating the need for the kernel to copy data to and from kernel buffers.

Threads provide several advantages over conventional UNIX style processes. The kernel can create threads, terminate threads, and context switch between threads much faster than systems with conventional processes. Because address spaces specify the mapping from virtual addresses to physical addresses, the kernel only needs to initialize the thread's state information and map in its stack region at thread creation time. All threads in an address space efficiently share the data in the address space and immediately reflect any changes made to the data. Threads also provide the ability to concurrently execute multiple tasks within the operating system kernel. Many kernel operations such as page reclamation, network management, and background paging are coded simply and elegantly when viewed as concurrent kernel threads. Section 4.5 describes how the virtual memory system uses kernel threads to efficiently support remote paging.

4.5 Virtual Memory Management

In VM Xinu, the virtual memory system uses all the design decisions discussed up to this point to efficiently support remote memory backing storage. First, the virtual memory system resides above the file system, networking, and device driver layers in the hierarchical design, permitting the virtual memory system to use a wide range of backing storage mediums. Second, the virtual memory system uses the abstract virtual memory machine to access the underlying hardware. Third, the virtual memory system uses lightweight kernel threads to communicate across the communication channel with the remote memory server. Because the threads execute in the same address space, they efficiently communicate with each other using shared memory.

Figure 4.5 shows the basic components of the virtual memory system and illustrates the relationships between the components. To briefly outline how the virtual memory system operates, imagine that a user-level application references a page that has been stored on the backing store. The hardware detects the access, raises a page fault exception, and begins executing a page fault handler routine. The page fault handler routine enqueues a page fetch request with a special purpose kernel thread whose sole purpose is to transmit paging requests to the memory server. The memory server responds with a page fetch reply which is received by another special purpose kernel thread that constantly listens for reply messages. The receiving thread then informs the user application (that caused the page fault handler be invoked) that the missing page has arrived.

4.5.1 Page Replacement

To amortize the cost of page replacement over time, a lightweight kernel thread, called the *frame manager*, periodically executes the replacement algorithm in the background, maintaining a constant supply of free frames for future physical memory requests. The operating system associates a low and high water mark with the free

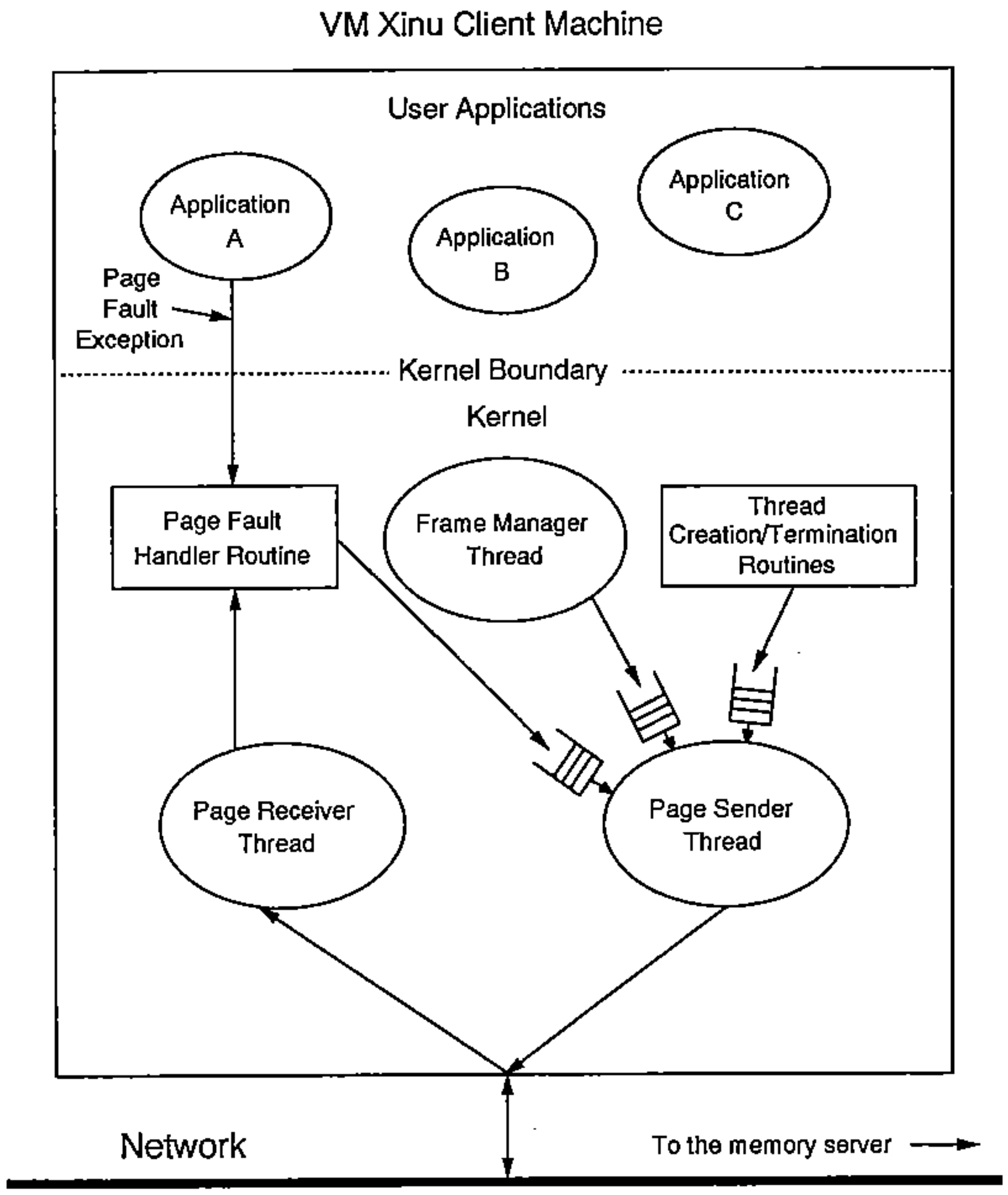

Figure 4.5 The organization of the virtual memory system in VM Xinu.

list to regulate the amount of time the frame manager executes. When the supply of available frames falls below the low water mark, the frame manager begins running in the background until the supply of free frames rises above the high water mark. The low water mark indicates a dangerously small number of available frames, while the high water mark represents an adequate number of available frames. Normally, the frame manager runs periodically for a fixed amount of time, keeping the supply of available frames near the high water mark. When the number of free frames becomes dangerously low, the frame manager runs continuously until the supply becomes adequate again. [3]

VM Xinu uses a modified version of the two-handed clock algorithm to approximate the LRU global replacement policy. The operating system maintains three lists to enhance the replacement algorithm: an *active list*, a *modified list*, and a *free list*. The active list contains all physical pages currently in use. The modified list contains modified pages waiting to be written to the backing store, and the free list contains unused pages and reclaimed pages. The two-handed global clock algorithm traverses the active list searching for pages to replace. When the algorithm locates a page for replacement, it moves the page to the modified list or to the free list. When an application requests physical memory, the operating system removes pages from the free list and assigns them to the application. The modified list and the free list give pages a *second chance*, increasing the *not referenced* time period and the probability of choosing the least recently used page. If an application accesses a page on the modified list or the free list, the page fault handler returns the page to the active list.

The low-level physical memory layer knows nothing about the page replacement algorithm. Low-level physical memory allocation routines obtain physical frames from the free list and block when no frames are available. Only the frame manager thread executes the page replacement algorithm to search for frames.

[3]Choosing the optimal values for the high-water mark and the low-water mark are outside the scope of this research. Consequently, VM Xinu simply sets the high-water mark and the low-water mark to a predefined static value.

The kernel maintains information about each physical frame in a data structure called the *frame table.* The frame table contains one entry for each page of physical memory. The operating system links each frame table entry into one of the three lists: active, modified, or referenced. Locked frames, such as frames allocated to kernel text and data, always reside in memory and do not appear on any of the lists or participate in the replacement algorithm.

4.5.2 Paging

Two lightweight kernel threads handle all the paging related I/O. The *page sender* thread handles all outgoing messages (i.e., paging requests), and the *page receiver* thread handles all incoming messages (i.e, paging replies). The frame manager never transmits paging requests to the memory server, and the page fault handler never receives paging replies from the memory server. Instead, the frame manager and page fault handler request these services from the page sender and page receiver threads. When the frame manager wants to store a page on the memory server, the frame manager enqueues the page with the page sender thread. The page sender thread formulates the request message and then transmits the message to the memory server. Similarly, the page fault handler enqueues a fetch request with the page sender thread and then waits for the page receiver thread to notify it that the desired page has arrived. The page sender and page receiver threads use interprocess communication and shared data to communicate quickly and efficiently.

The page sender thread maintains three distinct queues for paging requests: a creation/termination queue, a page-in queue, and a page-out queue (see Figure 4.5).[4] The frame manager, page fault handler, and creation/termination routines invoke the page sender by sending a request to one of the three queues. The page sender wakes up, services the enqueued requests, and then waits for more requests.

When a reply from the remote memory server arrives, the operating system awakens the page receiver thread to process the reply. Page-in replies cause the page

[4]Chapter 5 discusses the various paging request types in greater detail.

receiver thread to awaken the faulting process after placing the page in the process' address space. When the page receiver thread receives a page-out reply, it moves the corresponding physical frame from the modified list to the free list. If a page-out error occurs, the page receiver resends the page-out message by enqueuing the message with the page sender.

Because the page sender and page receiver threads handle all backing store I/O, the virtual memory system architecture can be easily modified to use a conventional magnetic disk for backing storage instead of remote memory. Only the page sender and page receiver threads must be modified to read and write data to and from a local disk. The remainder of the virtual memory system architecture requires no changes to convert to conventional magnetic disk storage or other forms of backing storage.

4.5.3 Memory Reclamation

When a thread or address space terminates, the operating system may consume a substantial amount of processing time reclaiming the memory allocated to the teminated thread or address space. To amortize the cost of memory reclamation over time, VM Xinu uses a mechanism called *timestamps.* The operating system associates a timestamp with every thread. When the operating system allocates a physical frame to a thread, the allocation routines stamp the frame with the owner's ID and timestamp. As long as the frame's timestamp matches the owner's timestamp, the operating system knows the frame is valid. When a thread terminates, the operating system must reclaim all the physical pages owned by the thread. To reduce the time required to terminate a thread, the kernel assigns a new timestamp to the thread,[5] thereby invalidating all physical frames owned by the thread. Consequently, the kernel terminates threads in constant time because it does not spend any time searching lists or freeing physical frames. Instead, the frame manager thread reclaims these pages in the background over time. As the frame manager scans the active list using the

[5]VM Xinu uses the lifetime of an address space/thread as the basic time unit. Consequently, a new timestamp is obtained by incrementing the old timestamp.

two-handed clock algorithm, the clock hands check each frame's timestamp against the owner's timestamp before processing the frame. If the timestamps do not match, the frame manager frees the frame and moves it to the head of the free list. The operating system uses the same algorithm to terminate address spaces. As a result, the kernel can terminate address spaces and threads in constant time.

4.6 Related Work

Relatively few hierarchically-designed operating systems exist. Dijkstra's THE operating system [Dij68] used a hierarchical design; however, it did not incorporate virtual memory into the design.[6] Several object-oriented systems, such as the the Choices operating system and the DASH kernel, encapsulate operations and data structures into objects with well-defined interfaces but do not specify the relationship or dependencies between objects [Rus91, AF88].

The Mach operating system [Ras86, YTR87, Tev87, BBB$^+$87] employs a scheme similar to VM Xinu to provide architecture independence. Mach divides the implementation into a machine-dependent component and a machine-independent component. Mach draws the line between machine-dependent and machine-independent code at a much higher level, resulting in a much larger machine-dependent component than VM Xinu's abstract machine component. Each time the system is ported to a new architecture, the large machine-dependent component must be rewritten. High-level operations that are architecture-independent in VM Xinu, such as zero-filling a physical page, copying a physical page to another physical page, removing all references to a physical page, or marking all references to a physical page as copy-on-write, are machine-dependent in Mach and require elaborate data structure support [Tev87]. In addition, Mach's machine-independent component keeps a redundant machine-independent copy of all virtual address space mappings, duplicating information and introducing coherency problems.

[6]The THE system did use a drum for backing storage; however, the hardware did not provide any virtual memory support, so "paging" was painfully obvious to the programmer.

The UNIX operating system [Bac86, LKKQ89, GMS88] provides portable memory management by assuming all page tables reside in memory directly addressable to the kernel. In-memory page tables map nicely onto the VAX architecture where the UNIX operating system has its roots. In fact, many versions of UNIX use the VAX architecture as an abstract machine and mimic the VAX architecture rather than exploit the functionality provided by the memory management unit.

The SunOS, Sprite, and Plan 9 operating systems support diskless nodes that page across a network to a remote file server as well as nodes that page to a local disk [GMS88, Sun90, OCD+87, Nel86, PPTT90]. Because the remote file system provides the same services that the local file system provides, the virtual memory system requires no changes because it is completely unaware of the location of the data.

Apollo's distributed DOMAIN operating system [LLD+83] provides an object-based model of computing where portions of an object may be migrated to another node. The system is similar to VM Xinu in the sense that the migrated portions of the object are accessed across a network. To access an object, the virtual memory system invokes a *locating service* that executes a distributed look-up algorithm to locate the object. The virtual memory system then uses a hardware-specific communication protocol to access the object across a ring network.

The Mach operating system uses a local disk as the default backing store. However, Mach allows users to create their own *external pagers* to provide backing storage for portions of the user's virtual address space. Using this mechanism, users could implement external pagers capable of accessing remote memory backing storage.

4.7 Summary

This chapter describes the design of an architecture-independent virtual memory operating system with support for remote memory backing store. The system uses a hierarchical design to clarify the relationships and dependencies between the various system components. At the lowest level of the hierarchy, the architecture interface layer implements an abstract virtual memory machine. The virtual memory system builds on the abstract virtual memory machine and requires no modifications when ported to new architectures. The paging system occurs at a high level in the hierarchy. Consequently, the paging system can use the file system, network, or local devices to access a wide range of backing storage mediums.

The kernel supports multiple threads of control within the kernel in addition to multi-threaded user applications. The virtual memory system uses two kernel threads, the *page sender thread* and the *page receiver thread*, to transfer data across the network to the remote memory backing store. A third kernel thread, the *frame manager thread*, executes the page replacement algorithm and communicates with the page sender thread using shared memory and efficient interprocess communication.

Finally, the virtual memory system uses a delayed page reclamation mechanism to terminate processes in constant time. When the virtual memory system allocates a physical page, it stamps the page with the current owner's timestamp. When the owner terminates, the virtual memory system increments the owner's timestamp and effectively invalidates the page. Because the frame manager thread reclaims these obsolete pages in the background, the cost of page reclamation is amortized over time.

5. A HIGH-SPEED REMOTE MEMORY COMMUNICATION PROTOCOL

Chapter 3 described the remote memory model which consisted of one or more memory servers and multiple client machines interconnected by a communication channel. Chapter 4 examined the client side of the model and described a hierarchical, portable, virtual memory operating system with support for remote paging. This chapter focuses on the communication channel and the protocol clients use to access remote memory.

In the remote memory model, clients use the communication channel to access the additional memory storage space provided by the memory server. Because the efficiency of the communication protocol significantly impacts the performance of the entire system, we designed a special purpose communication protocol called the *Remote Memory Communication Protocol* (RMCP). Clients use the protocol to transfer data to the memory server when they need additional memory space, and the memory server uses the protocol to transfer data back to the client when the client requests the data. The remote memory communication protocol hides the characteristics of the underlying communication channel from the client's virtual memory system and the remote memory server. The protocol allows clients to communicate with memory servers over almost any network hardware.

The remote memory model defines a conceptual model for designing distributed systems, not a specific implementation, and intentionally leaves the details unspecified. In particular, the remote memory model does not specify the characteristics of the communication channel. Before we can design a communication protocol we must establish a minimal set of characteristics or assumptions regarding the communication channel and the machines it connects. To narrow the possible characteristics of the communication channel we make the following assumptions:

Connectivity: The communication channel provides message delivery between a memory server and each client machine. If necessary, the communication channel handles routing (e.g., across gateways) to provide a transmission path between the server and a client.

Datagram Service: The communication channel provides datagram (packet) oriented service.

Unreliable Delivery: The communication channel unreliably delivers packets. The communication channel may drop, corrupt, or delay packets. However, the rate at which these errors occur is relatively low.

High-Speed: The communication channel offers a relatively high bandwidth and low delay (e.g., a 10Mbps Ethernet or FDDI).

Heterogeneous Clients: The distributed system consists of heterogeneous client architectures and operating systems.

Concurrent Server Access: The memory server serves multiple client machines simultaneously, providing concurrent access to the remote memory resource.

This small set of assumptions serves as the basis for the design of the Remote Memory Communication Protocol. The following section outlines our goals for the Remote Memory Communication Protocol and the effect our assumptions have on the design of the protocol.

5.1 Design Goals

Clearly, the communication protocol's performance significantly influences the virtual memory system's performance. To improve the performance of the virtual memory system, the communication protocol should provide highly efficient data transfer (i.e., minimize the time required to transfer data between a client and the memory server). In addition to *efficiency*, the communication protocol should provide *reliability* and *architecture independence.*

5.1.1 Reliability

When a client exceeds the capacity of its local memory, the virtual memory system selects one or more pages of memory and transfers the data stored in those pages to the backing store. Only after the virtual memory system's I/O mechanism has successfully transferred and stored the data on the backing store can the virtual memory system reclaim the memory occupied by the data. That is, to insure the virtual memory system operates correctly, the virtual memory system's I/O mechanism must be able to obtain the status of every I/O operation issued to the backing store. If an operation fails, the virtual memory system must repeatedly issue the operation until it succeeds. If a store operation fails and the virtual memory system does not detect the failure, the virtual memory system may try to retrieve the data and find it missing. Similarly, the virtual memory system cannot allow a process to access a page of memory until the fetch operation succeeds.

In addition, the backing store must execute store requests and fetch requests in the order the virtual memory system issues them. For local disk backing storage, this requirement does not present a problem because the disk only processes one operation at a time. However, for remote memory backing storage accessed across a network, the virtual memory system may issue multiple, concurrent, paging operations. Because the network drops messages and delivers messages out of order, the I/O mechanism must guarantee that the paging operations will be processed in order. If the remote memory server executes the paging requests out of order, the client's virtual memory system may fail.

In the remote memory model, the communication protocol handles all I/O between clients and the memory server. Consequently, the communication protocol must guarantee reliable, in-order processing of all paging operations.

5.1.2 Architecture Independence

The recent proliferation of network architectures and computer architectures, combined with a desire to execute the communication protocol across a wide variety of hardware platforms, mandates a communication protocol design that limits the number of modifications required to port the protocol to a new architecture. In particular, the communication protocol must exhibit independence from the underlying network architecture, the client architectures, and the server machine's architecture.

Instead of binding the design of the communication protocol to a particular network architecture, the communication protocol should provide the flexibility required to run on an assortment of network technologies. Limiting the number of assumptions about the communication channel facilitates the design of an abstract communication protocol that maps easily onto a wide variety of network architectures. Consequently, the only assumption the communication protocol makes about the underlying communication channel is that the communication channel provides unreliable datagram service between every client and the memory server. Many conventional physical network architectures (e.g., Ethernet, proNET, FDDI, or point-to-point networks) meet this minimal constraint as well as several virtual network architectures (e.g., TCP/IP networks such as the Internet).

Because the distributed system consists of heterogeneous client machines, the page size of each client will differ from machine to machine. Consequently, the protocol must support variable size messages, allowing heterogeneous client architectures to send and receive architecture-dependent size pages to and from the memory server. To complicate matters, a client's page size, and therefore the message size, may exceed the MTU of the underlying communication channel. To be truly portable, the communication protocol must transfer messages of arbitrary size, regardless of the underlying network architecture's MTU.

Finally, the communication protocol should adhere to the network protocol layering principle [Com88]. The layering principle says that for any layer n in a hierarchy of network layers, the destination's layer n receives exactly the same message sent by layer n at the source. This means that the receiver must receive a message identical to the message sent by the sender.

5.1.3 Efficiency

The virtual memory operating system described in the previous chapter provides support for demand paging. When a process accesses a page that has been written to backing storage, the virtual memory system suspends execution of the process until it can retrieve the page from the backing store. Retrieving a page from backing storage involves locating a page of physical memory and (possibly) writing the contents of the page to the backing store before retrieving the faulting page's data. Because the time to access local memory and the time to access backing storage differ by several orders of magnitude, the suspended process experiences a sizeable delay when compared to a local memory access. In addition, the system must block processes waiting for resources held by the suspended process. The system must also block all processes waiting for output from the suspended process. Consequently, to improve the virtual memory system's performance, the communication protocol should minimize the time required to process a store or fetch request from a client. To improve efficiency the communication protocol should:

- minimize the number of messages sent between a client and a server when executing a store or fetch request,
- minimize the number of packets used to transmit a message across the communication channel,
- minimize the per message processing overhead on the client and the server (e.g., processing the protocol headers),
- take advantage of the asynchronous nature of the underlying communication channel whenever possible by issuing multiple, concurrent, requests.

Unfortunately, the reliability requirements mentioned in section 5.1.1 often result in additional messages and added overhead, reducing the efficiency of the protocol. The tradeoff between reliability and efficiency makes it difficult to find the correct balance that will optimize overall system performance. Because we assume the underlying communication channel has a relatively low error rate, the protocol optimizes performance for the most common case; the case when no errors occur.

5.2 Conceptual View Of The RMCP Protocol

The *Remote Memory Communication Protocol* (RMCP) is a special purpose protocol designed to meet the reliability and architecture independence requirements described in section 5.1. The protocol provides reliable delivery, data streaming, arbitrarily large messages that permit arbitrary page size transfers, independence from the underlying network architecture, and low overhead that results in high efficiency. All client machines use RMCP to transfer data to and from the memory server.

The Remote Memory Communication Protocol consists of two layers: the *Xinu Paging Protocol* (XPP) Layer and the *Negative Acknowledgement Fragmentation Protocol* (NAFP) layer. Figure 5.1 illustrates the layering and the flow of data through the layers.

Dividing the protocol into two distinct layers clearly separates the tasks performed by the protocol into high-level *paging* operations and low-level *transport* operations. The XPP layer implements the high-level abstract paging operations used to reliably store and retrieve data to and from the memory server. The NAFP layer implements the low-level transport operations required to transfer XPP messages across the communication channel. NAFP hides the characteristics of the communication channel and the details of message delivery from the XPP layer, allowing the XPP layer to remain independent from the underlying network architecture.

Both the XPP layer and the NAFP layer adhere to the network protocol layering principle. That is, the XPP layer on the server side receives messages identical to the messages sent by the client's XPP layer, and vice versa. Similarly, the server's NAFP layer receives an exact copy of the datagrams sent by the client's NAFP layer.

Because the Remote Memory Communication Protocol imposes minimal requirements on the communication channel, the protocol executes over any transport mechanism that provides unreliable datagram service. Almost any hardware or software

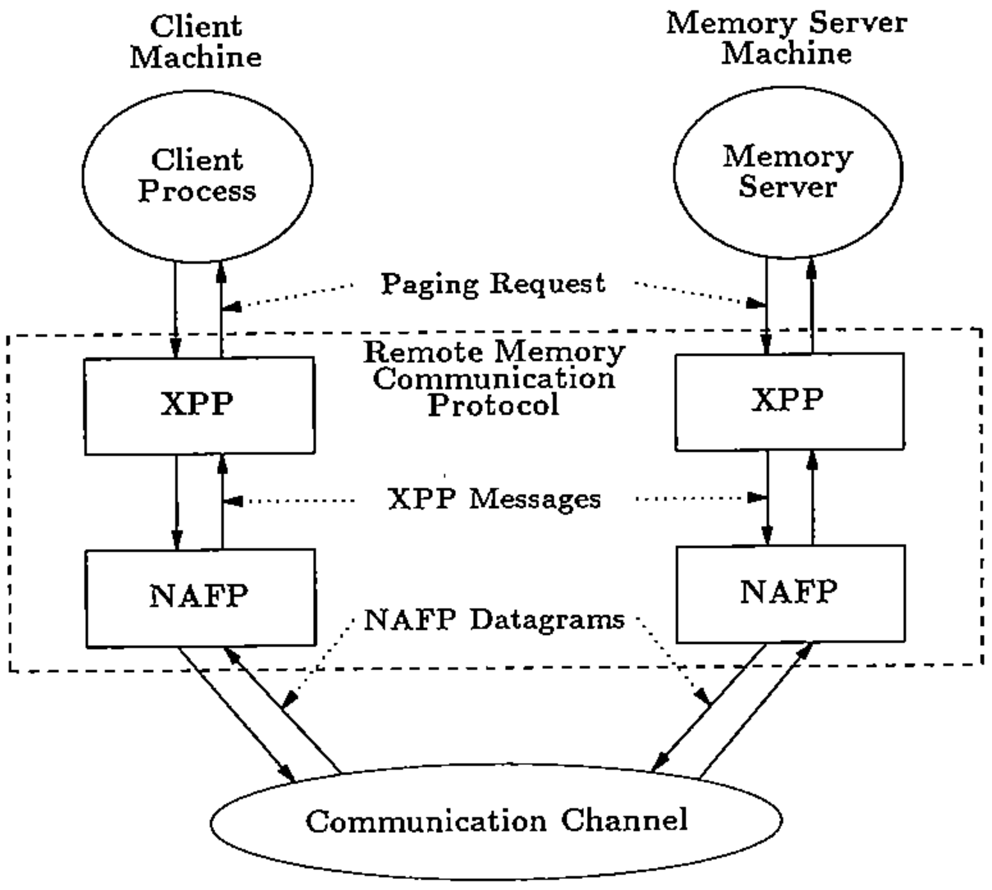

Figure 5.1 The two layers of the remote memory communication protocol. A client initiates an XPP request which travels down through the layers, across the communication channel, and back up through the layers to the memory server. The memory server processes the request and sends an XPP reply back through the layers and across the communication channel to the waiting client process.

transport service can function as the underlying communication channel, including transport mechanisms that provide reliable datagram service, stream service, or virtual circuits.

RMCP, as a whole, functions much like a remote procedure call [BN84, Lyo84, Wel86].[1] Assuming no errors arise, the protocol operates as follows: A process on a client machine desires to store or fetch a page of memory to or from the memory server. The client process invokes the XPP layer passing it the desired request and all pertinent data. The XPP layer creates an appropriate message, passes the message to the NAFP layer for transmission, and waits for a response from the server. The NAFP layer transfers the message across the communication channel, possibly fragmenting the message and reassembling it on the server side. Then, the NAFP layer gives the message to the XPP layer which decodes the message and passes the message type and associated data to the memory server. The memory server processes the request and sends a reply message containing the results down through the layers to the communication channel, across the channel, and then back up through the client layers. When the client XPP layer receives the reply, it informs the caller that the requested operation has successfully completed and passes any results the operation may have produced to the caller.

5.3 The XPP Protocol

The Xinu Paging Protocol provides the virtual memory system with the ability to issue high-level abstract paging operations. XPP hides the details of backing store I/O from the virtual memory system by reliably transmitting paging messages, invoking the desired paging operation on the memory server, and returning the results. The XPP protocol packages up the requested operation into an XPP message and invokes NAFP to deliver the message across the communication channel to the memory server. The XPP layer on the server unpacks the XPP message to obtain the original operation and delivers the requested paging operation to the memory server. When

[1]We discuss the similarities/differences to remote procedure calls in section 5.5.

the server completes the operation, it sends an XPP reply message to the client to inform the virtual memory system that the operation has finished. If the NAFP layer fails to deliver an XPP request message, the XPP layer will retransmit the message until it succeeds. Because the XPP protocol provides reliability, the virtual memory system only needs to issue a store page or fetch page operation once and wait for the results.

5.3.1 XPP Message Types

5.3.1.1 Conceptual Message Types

The XPP protocol supports four conceptual message types which the virtual memory system uses to access the memory server. Each message type corresponds to one of the memory server's abstract paging operations.[2] Four abstract paging operations provide the functionality needed to support the virtual memory system. These four basic message types are:

- page store requests
- page fetch requests
- process create requests
- process terminate requests.

The XPP protocol assumes that each client's virtual memory system divides memory into *pages*. Consequently, XPP uses a *page* as the basic unit of data transfer between the client and the server. Although XPP makes this assumption, it does not specify the size of a page. Each client independently choses a page size that suits the client. Usually the client choses a page size that matches the underlying architecture's page size.

The client's virtual memory system uses page store request messages and page fetch request messages to send and receive data to and from the server. When a

[2]Chapter 6 describes the operations provided by the memory server.

client creates or terminates a process, the virtual memory system informs the memory server using a process create request or a process terminate request. We discuss create and terminate requests in greater detail in section 6.3.4. For the moment, suffice it to say that the server uses this information to provide efficient storage and to detect erroneous paging requests.

5.3.1.2 Concrete Message Types

The four conceptual XPP message types correspond to high-level paging operations and provide the virtual memory system with a high-level interface to the remote memory backing store. However, XPP uses several message types to implement the conceptual message types and provide reliability, data streaming, and error detection.

A list of the message types supported by XPP, along with a brief description of each type, appears in Table 5.1. The first eight message types implement the four conceptual message types used by the virtual memory system. Each conceptual message type corresponds to a pair of XPP messages: an XPP request message and an XPP reply message. The reply message serves two purposes: it acknowledges receipt of the request message, and it returns the results of the requested operation. In the case of a *fetch_request*, the reply message contains the requested data. In all other cases, the reply contains the status of the requested operation.

XPP establishes contact with a memory server using an XPP *startup_request* message. Before sending any other messages, XPP issues a startup_request message to request permission from the memory server to use it for backing storage. The memory server then issues an XPP *startup_reply* message to grant or deny permission. If an error occurs, the memory server reports the error with an XPP *illegal_request* message. For example, when a client attempts to fetch a page that does not exist, the server signals the error with a illegal_request message.

The XPP protocol includes two additional messages to improve efficiency: a *shutdown_request*, and a *create-and-store_request*. The *shutdown_request* message functions

Table 5.1 XPP message types.

Message Type	Meaning
STORE_REQUEST	Store data on the memory server
STORE_REPLY	Memory server successfully stored the data
FETCH_REQUEST	Retrieve data from the memory server
FETCH_REPLY	Requested data enclosed in message
CREATE_REQUEST	Inform the server of a new process
CREATE_REPLY	Create_request acknowledgement
TERMINATE_REQUEST	Inform the server of a terminated process
TERMINATE_REPLY	Terminate_request acknowledgement
STARTUP_REQUEST	Client system boot request
STARTUP_REPLY	Startup_request acknowledgement
SHUTDOWN_REQUEST	Client system shutdown request
SHUTDOWN_REPLY	Shutdown_request acknowledgement
CREATE_STORE_REQUEST	Create process and store page
CREATE_STORE_REPLY	Create_Store_request acknowledgement
ILLEGAL_REQUEST	Error: Illegal request

as the counterpart to the startup_request message. If a client system terminates gracefully, it sends a shutdown_request message to the memory server to inform the server that it no longer requires backing storage service. The server uses this information to release memory server resources held by the client. Because clients may go down unexpectedly, the server does not require notification via a shutdown message.[3]

XPP also supports a *create-and-store_request* that combines a *create_request* and a *store_request* into a single message. Create-and-store_requests allow a client to delay sending the create_request for a process until the virtual memory system issues a store_request for the process. The client then sends a single message instead of two separate messages. Thus, if a process never stores any data on the memory server, the client machine does not send any requests to the server.

5.3.2 Reliability

The XPP protocol satisfies the end-to-end reliability requirement imposed on the RMCP protocol in section 5.1.1. Many conventional protocols and network architectures provide *reliable delivery* [Pos81, Org86, Che88, Dig84]. Protocols that provide reliable delivery accept messages from a source host and guarantee delivery of the messages to the specified destination host. However, a virtual memory system requires more than reliable delivery from the communication protocol.

Protocols that provide reliable delivery make no promises about the handling or treatment of a message once the message has been received (and acknowledged) by the receiver. In particular, a protocol that only provides reliable delivery cannot guarantee that the memory server will process a paging request once the request has been delivered. However, guaranteeing that the server will process the request is precisely the guarantee the client virtual memory system requires. That is, the client's virtual memory system requires an *end-to-end* [SRC84] reliability guarantee from the communication protocol. The virtual memory system only wants to know whether the server successfully or unsuccessfully processed the request. The virtual

[3]Chapter 6 discusses the server's actions for startup and shutdown messages in greater detail.

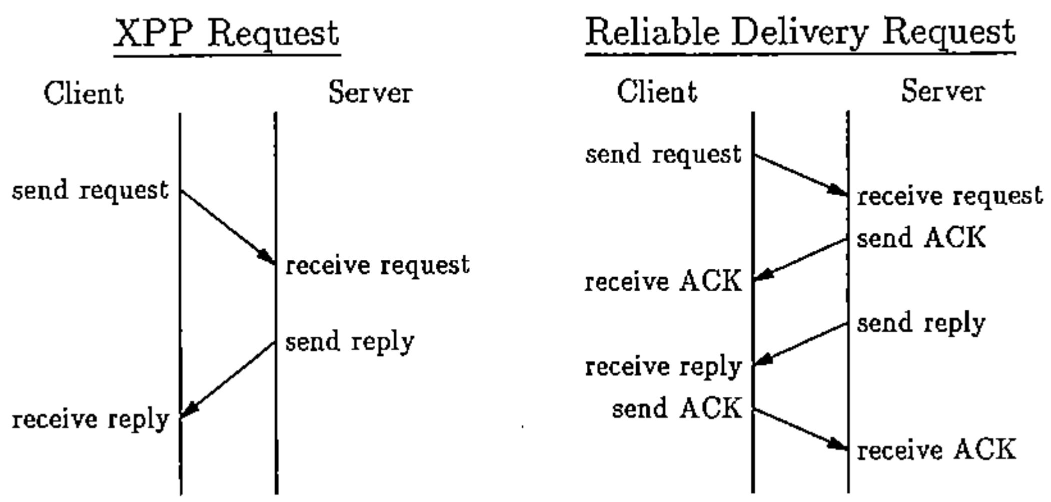

Figure 5.2 Protocols that guarantee reliable delivery use 4 messages to process a request. XPP only uses 2 messages for most paging requests.

memory system expects the I/O mechanism (i.e., the communication protocol) to reliably perform the paging operation specified in the request message. To provide this type of guarantee, the protocol may have to transmit a paging request multiple times. Even if the protocol knows that the server received the request, the protocol must resend the request message until it receives a reply message.

Protocols that guarantee reliable delivery typically use four messages per request. XPP does not waste any effort trying to reliably deliver individual messages that may have to be resent even though they were received by the server. To improve efficiency. XPP only uses two messages for most paging requests: a *request* message followed by a *reply* message (see Figure 5.2). XPP does not try to reliably deliver either message. If the client does not receive the reply message in a timely fashion, the client assumes an error occurred and resends the request. As long as no errors occur, each paging operation results in only two XPP messages.

XPP uses timeouts, retransmissions, and positive acknowledgements to provide end-to-end reliability. Positive acknowledgements, in XPP, function differently than positive acknowledgements in protocols that guarantee reliable delivery. Positive acknowledgements in protocols that guarantee reliable delivery indicate that the destination received the message [Com88]. In XPP, the reply to a request message acts as a positive acknowledgement. In addition to acknowledging receipt of the request message, a reply message indicates that the request message was processed by returning the status of the requested operation. XPP starts a timer when it sends a request message and retransmits the request message if the timer expires before the reply message arrives. XPP saves a copy of the message and does not delete the request until it receives a successful reply. Every time XPP resends the request message, it resets the timer and waits for a reply.

The XPP layer on the server functions differently than the XPP layer on the client, due to the request-reply nature of XPP. Because multiple client machines simultaneously access the memory server, the server can quickly become a bottleneck, degrading the performance of all the client virtual memory systems. To simplify the server, the XPP layer on the server does not reliably deliver reply messages. The server knows the client will reissue the request if the reply message does not successfully reach the client. Consequently, the XPP layer on the server does not waste any effort trying to reliably deliver the reply message. The client driven nature of the XPP protocol simplifies the memory server and optimizes performance for the expected case; the case in which the client receives the reply. The server does not need to implement timers, save messages, wait for acknowledgements, or retransmit messages. As a result, the server spends more time processing client request messages and less time communicating.

5.3.3 Message Sequencing

Virtual memory systems, including VM Xinu's virtual memory system, expect paging requests to be processed in the same order that the requests were issued. Consequently, XPP must guarantee to deliver paging messages to the memory server in-order. If the server does not process the messages in the order they were issued, the server may return incorrect data in response to a fetch request.

XPP uses sequence numbers to guarantee that request messages are delivered in the correct order. The client assigns a sequence number to each XPP message it sends to the server. The client and the server agree on an initial sequence number and increment the sequence number for each new request message sent. Each XPP client/server pair has its own sequence number.

A sequence number serves two purposes: it uniquely identifies a message, and it imposes an ordering on the list of messages. When the server returns an operation's results in an XPP reply message, the server includes the sequence number from the corresponding request message. The client identifies the reply message by its sequence number and quickly locates the corresponding request message with a matching sequence number. Because XPP retransmits messages, the server and the client may receive duplicate messages. The sequence numbers allow XPP to detect duplicate messages and take the appropriate action.[4]

Because XPP assigns sequence numbers in increasing order, the sequence numbers impose an ordering on the list of messages. XPP uses sequence numbers to deliver messages to the server in an order that insures correct results. However, the virtual memory system does not need to impose such a strict ordering to achieve correct results. In practice, the virtual memory system only needs to impose a partial ordering on the list of messages to achieve correctness. That is, the case seldom occurs where the server must process message i before processing message $i+1$. More often the case arises where the server can process message $i+1$ before message i and still achieve the same results. For example, assume a client issues a store request for physical page i belonging to process A followed by a fetch request for physical page j belonging to process B. Because the two messages do not depend on each other in any way, XPP may deliver the messages out of order without affecting the result. Consequently, XPP will, under certain circumstances, pass out of order messages on to the server in an attempt to improve efficiency.

[4]The appropriate action depends on the type of message. For example, when a memory server receives a duplicate fetch_request, it sends a reply containing the requested page. However, when a client receives a duplicate fetch_reply, it discards the message.

Each XPP message includes a *preceding message number* which XPP uses to impose a partial ordering on the list of messages. We say that message j depends on message i if the server must process message i before processing message j. We define the relation $\preceq$ on the set of messages as

$$m_i \preceq m_j \text{ if } m_j \text{ depends on } m_i \text{, or if } i = j \ .$$

The relation $\preceq$ defines a partial ordering on the list of messages $m_1, m_2, ..., m_n$. XPP remembers the last sequence number the server processed. When the XPP layer on the server receives a new request, it checks the sequence number against the last sequence number to see if the server missed any request messages. If XPP receives a request message out of order, it uses the partial ordering to determine whether the server should process the out of order message anyway or wait until the missing request arrives.

Because clients issue messages with increasing sequence numbers, m_i does not depend on m_j for $i < j$. Consequently, for a given message m_j, XPP only needs to verify that

$$m_i \preceq m_j \text{ is false for all } i, \ l < i < j \ .$$

where l is the last sequence number processed. XPP uses the preceding message number to perform the verification in a single operation. We define the preceding message number as

$$p = \ \max i \text{, such that } i < curmsg \text{ and } m_i \preceq m_{curmsg} \ .$$

That is, the preceding message number specifies the sequence number of the most recent preceding message that must be processed before the current message can be processed. Because the client virtual memory system knows the type and content of each XPP request message, it computes and sends a preceding message number with each XPP message. XPP compares the preceding message number (p) with the sequence number of the last message the server processed (l). If $p \leq l$, XPP gives the message to the server for processing rather than allowing the server to sit idle waiting for the missing message(s).

The concept of a preceding message number is also applicable to other protocols where there exists a partial ordering on the list of messages. For example, imagine a network windowing system protocol (e.g., a protocol similar to the X or NeWS protocols[Nye90, Sun]) in which a partial ordering exists on the list of screen manipulation messages. Preceding message numbers would allow the system to process independent screen manipulation messages out of order (e.g., two independent draw_line messages).

5.3.4 Data Streaming

XPP uses data streaming to increase concurrency and total throughput. Instead of sending one message at a time, XPP allows the virtual memory system to issue several XPP messages concurrently. To use as much of the network bandwidth as possible, XPP sends a stream of messages to the server.

Allowing the virtual memory system to issue more than one XPP message at a time increases the paging throughput by increasing the level of concurrent processing. We can view the steps an XPP message takes on its way to the server and back as a multistage pipeline with distinct processing entities executing the various stages in parallel. Figure 5.3 illustrates the steps an XPP message goes through and the three processing entities that execute each stage of the pipeline. Because XPP allows the virtual memory system to asynchronously send multiple page requests to the server, XPP keeps the pipeline full and increases the throughput. The client creates a new message while the communication channel transfers the previous message. At the same time, the server processes the first message sent through the pipeline. Because all three entities execute simultaneously, the overall throughput increases.

XPP maintains a *pending list* of request messages that the server has not yet processed or acknowledged with an XPP reply message. To avoid overrunning the server with request messages, XPP limits the length of the pending list, only allowing a finite number of outstanding requests at any given time. XPP sends multiple request messages back-to-back until it fills the pending list. The client then waits for

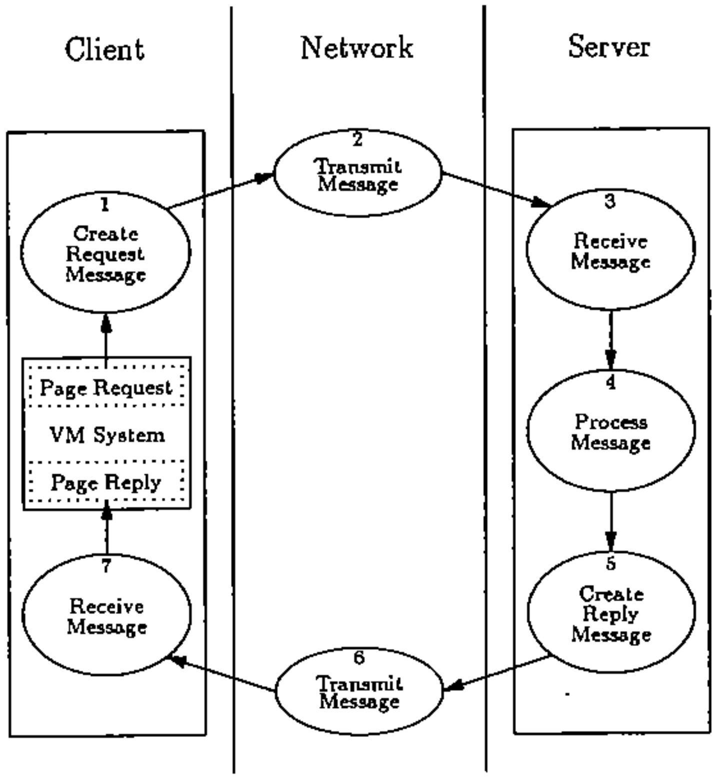

Figure 5.3 The stages of an XPP message. The client machine processes stages 1 and 7, the communication channel stages 2 and 6, and the server stages 3, 4, and 5. All three execute concurrently.

the server to process the messages and reply. When the client receives an XPP reply message corresponding to a request message on the pending list, the client removes the request message from the pending list and sends the next waiting request message.

Figure 5.4 compares data streaming to synchronous transmission and shows the effect of data streaming when using a pending list of length three. In the synchronous case, the paging system spends much of its time idle, waiting for messages. Data streaming, however, keeps the client and the server busy and increases throughput.

5.3.5 Message Format

An XPP message consists of the two components shown in Figure 5.5: an XPP header and an XPP data area. The XPP header contains information describing the contents of the message, while the XPP data area holds the data being transferred between the client and the server.

Figure 5.6 shows the fields that comprise an XPP message. XPP uses a fixed length header, but allows a variable length data area. The *TYPE* field specifies the XPP message type (e.g., *store_request* or *fetch_request*). Certain messages, such as the *illegal_request* message, use the *CODE* field to provide further information about the message type (e.g., illegal page fetch request).

Each XPP message contains a *MESSAGE NUMBER* used to identify the current message and a *PRECEDING MESSAGE NUMBER* that specifies the most recent message on which the current message depends. The *MACHINE ID*, *ADDRESS SPACE ID*, and *PAGE NUMBER* fields uniquely identify a page of data,[5] while the *PAGE SIZE* field specifies the length (in octets) of the data area. The ability to transfer variable size pages allows heterogeneous architectures to use a single protocol.

In the VM Xinu operating system two kernel threads implement the XPP protocol. The *page sender* and *page receiver* threads described in section 4.5 handle all paging requests issued by the virtual memory system. The page sender and page receiver

[5]Chapter 6 describes how the memory server uses these fields to uniquely identify memory regions on the memory server. In terms of the memory server these fields correspond to the LMS ID, VS ID, and Page Number (see section 6.2).

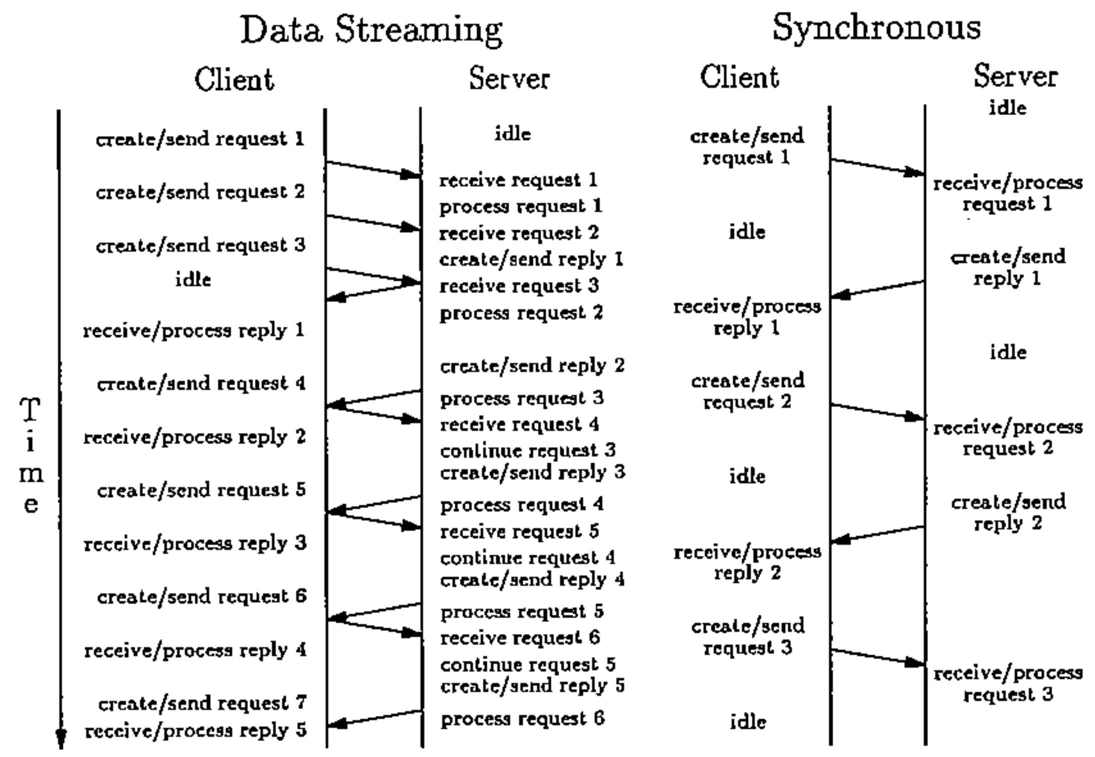

Figure 5.4 Data streaming vs. synchronous sends. Data streaming with a pending list of length 3 keeps the system busy, while synchronous sends result in a substantial amount of idle time. In each case, time moves down the page and shows the messages transferred.

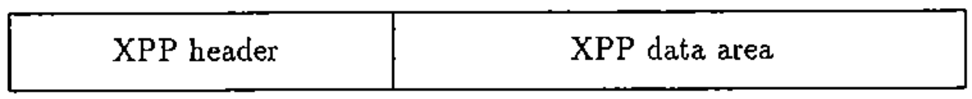

Figure 5.5 The two components of an XPP message.

threads work together to provide the reliability described earlier. The page sender thread sends all XPP request messages, and the page receiver thread receives all XPP reply messages. The page sender saves all unanswered request messages on the pending list for possible retransmission. It periodically traverses the list searching for requests that have timed out and resends them. The page receiver shares the pending list with the page sender and removes request messages from the pending list when the corresponding XPP reply message arrives.

VM Xinu's virtual memory system does not perform any I/O. When a page fault occurs, the virtual memory system asks the page sender thread (via interprocess communication) to issue a fetch request. Instead of creating a message, routing it, and transmitting it all at interrupt time, the virtual memory system lets the page sender handle the I/O. Likewise, the virtual memory system never listens on the communication channel. Instead, the page receiver listens on the communication channel and notifies the virtual memory system when an XPP reply message arrives.

To improve performance, the virtual memory system assigns priorities to each request type. The page sender transmits requests with the highest priority first. The virtual memory system enqueues requests on one of three queues at the page sender: a *page-in queue*, a *page-out queue*, and a *create-terminate queue*. To minimize the time between a page fault and resumption of the faulting thread, the virtual memory system assigns the highest priority to messages on the page-in queue. Because the virtual memory system sends pages to backing storage in the background, the page-out queue has the lowest priority. Internal to the page sender thread, retransmissions take highest priority. The virtual memory system knows the dependencies between requests and insures that any reordering resulting from the priority assignments will not violate the partial ordering on the list of messages.

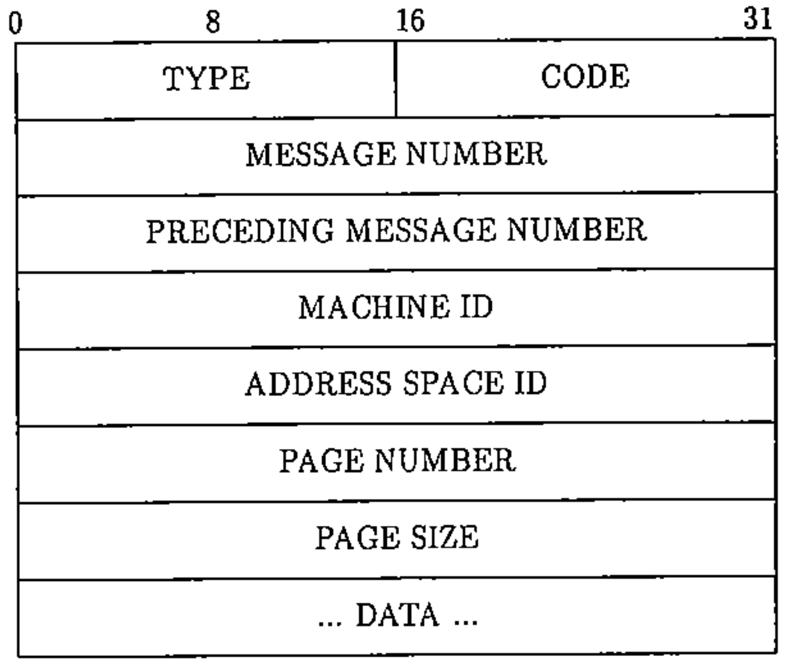

Figure 5.6 The fields in an XPP message. The first 8 fields make up the XPP header, and the remainder of the message comprises the XPP data area.

Finally, the page receiver acts as a dispatcher, demultiplexing incoming messages to the threads awaiting replies (i.e., threads blocked on a page fault). Similarly, because all messages go through the page sender, the page sender acts as a multiplexer. The ability to multiplex/demultiplex allows us to execute the XPP protocol over any physical or virtual communication channel that provides machine-to-machine message delivery (as opposed to requiring port-to-port or thread-to-thread delivery).

5.4 The NAFP Protocol

The Negative Acknowledgement Fragmentation Protocol (NAFP) provides the low level transport mechanism required to transfer XPP messages over the underlying communication channel. The NAFP layer accepts messages from the XPP layer, transfers the message across the underlying communication channel to the NAFP layer on the receiving side, and delivers the message to the receiving XPP layer. The NAFP layer hides the characteristics of the communication channel from the XPP layer, allowing the XPP layer to implement the high-level store and fetch messages in an architecture-independent fashion. NAFP provides fragmentation, message queueing, early error detection/correction, and an architecture-independent interface to the underlying communication channel. The result is a highly efficient data transport protocol.

5.4.1 Fragmentation and Reassembly

Because XPP supports a wide variety of client page sizes, the length of an XPP message may exceed the MTU of the underlying communication channel. The NAFP protocol hides the MTU of the communication channel from the XPP layer, allowing XPP to send and receive arbitrarily large messages carrying arbitrary size pages. NAFP accepts arbitrarily large XPP messages and breaks the messages into fragments that can be transmitted across the communication channel.

Each NAFP message consists of a header area and a data area. The header contains information about the NAFP packet type and identifies the contents of the data area. NAFP divides an XPP message into multiple fragments of equal size (except the last). The fragment size depends on the maximum size of an NAFP message. NAFP does not interpret the contents of the XPP message. Instead it treats the entire XPP message, header and data, as a contiguous set of uninterpreted bytes. NAFP encapsulates XPP fragments in the data portion of an NAFP message and transmits the fragments in order. Because NAFP does not treat the XPP header differently than the XPP data area, the XPP header only occurs in the first fragment; subsequent fragments only contain XPP data.

The NAFP header contains the information needed to reassemble the fragments on the receiver side. NAFP chooses a unique number called the NAFP message number to identify all the fragments of an XPP message.[6] Even though the XPP header only occurs in the first fragment, the NAFP message number indicates which fragments go together so that NAFP never needs to look at the data area. To insure that the receiver reassembles all the fragments in the correct order, the sender assigns a sequence number running from 1 to n to each fragment, where n is the total number of fragments making up the XPP message. The header also specifies the total number of fragments making up the XPP message so that the receiver will know when the last fragment has arrived and reassembly can begin.

The NAFP layer on the receiver maintains a list of buffers to reassemble incoming XPP messages. Each buffer contains a partially completed XPP message consisting of the fragments NAFP has received so far. When a fragment arrives, NAFP checks the message number and locates the buffer associated with the message number or allocates a new buffer if no buffer currently exists. Because XPP supports data streaming, NAFP may receive the fragments from multiple XPP messages simultaneously. Moreover, the NAFP layer on the memory server may receive multiple XPP messages from multiple clients simultaneously. Consequently, NAFP uses the message number and the client's network address to locate the proper buffer.

[6]NAFP usually uses the XPP message number as the unique identifier.

Because NAFP divides an XPP message into equal size fragments, the receiver, knowing the fragment size and the sequence number, can determine the exact location of the fragment within the buffer. Even if the fragments arrive out of order, the receiver will accept the fragment and place it at the correct position in the buffer. When all the fragments of an XPP message have arrived, the NAFP buffer contains a complete XPP message identical to the original message. Because XPP messages may arrive faster than the XPP layer can process them, the NAFP protocol enqueues incoming messages until the XPP layer has a chance to process them. When the XPP layer is ready to accept another message, it takes a message off the queue of incoming messages for processing. Because NAFP does not guarantee reliable delivery, NAFP drops incoming messages when the number of incoming messages exceeds the maximum length of the queue.

The maximum size of an NAFP message is based on the MTU of the underlying communication channel. That is, NAFP messages must fit in the datagrams provided by the communication channel. To reduce the number of datagrams NAFP uses to transfer an XPP message, NAFP uses a maximum packet size equal to the MTU of the communication channel. If NAFP runs over a virtual communication channel, however, the communication channel may perform its own fragmentation (e.g., at gateways) to provide a logical MTU much larger than the smallest MTU supported by the underlying physical networks. In this case, NAFP uses a maximum packet size equal to the smallest physical network MTU so that the communication channel will not fragment the NAFP packets. Because one of the goals of NAFP is to detect and correct fragmentation errors before they reach the XPP layer, NAFP does not want the communication channel to fragment messages. If NAFP, rather than the communication channel, performs fragmentation, then NAFP can detect fragmentation errors (e.g., lost fragments) and correct them. The following section describes how NAFP corrects fragmentation errors.

5.4.2 Negative Acknowledgements

Although conventional communication channels provide reasonably reliable packet delivery, they still drop packets, corrupt packet contents, deliver packets out of order, deliver packets after a substantial delay, or deliver duplicate packets. XPP combats these transmission errors by repeatedly sending each XPP request until the request succeeds. Although XPP will eventually recover from communication channel errors, it does not detect communication errors until some time long after the error occurs. When XPP finally does detect the error, recovering from the error can be expensive. For example, to send or receive an 8K byte page from a Sun 3 over an Ethernet with an MTU of approximately 1500 bytes requires a minimum of 6 packets. If the communication channel loses or corrupts any 1 of the 6 packets that make up the message, the message will not get through and XPP will not detect the error until the message times out. Then, XPP will resend all 6 fragments, even if only one fragment was lost.

The NAFP protocol uses *negative acknowledgements* (NACKs) to reduce the number of errors seen at the XPP level and improve the overall efficiency of the communication protocol. Conventional protocols that provide reliable delivery typically use positive acknowledgements, timeouts, and retransmissions to guarantee delivery. Because the XPP layer already guarantees reliability, NAFP should not provide reliable delivery of individual fragments, especially if the added reliability increases the overhead associated with NAFP. Instead, NAFP detects and corrects most, but not necessarily all, errors arising due to fragmentation but does not increase the delay substantially. That is, NAFP reduces the error rate seen by the XPP layer without degrading the performance of the transport protocol.

NAFP uses sequence numbers and negative acknowledgements (NACKs) to detect and correct fragmentation errors as soon as they occur. When NAFP receives an incoming fragment, it processes the fragment but does not acknowledge it. NAFP remembers the sequence number of the last fragment it received for each partially transmitted XPP message and uses the number to detect fragmentation errors. As soon as NAFP receives a fragment out of order (i.e., detects a missing sequence number), NAFP sends a negative acknowledgement to the sender's NAFP layer containing the missing fragment's sequence number. The sender's NAFP layer receives the NACK and immediately resends the missing fragment.

Because we assume the communication channel is unreliable, the NACK message may not reach the sender. NAFP does not guarantee reliable delivery of NACKs. Instead, NAFP only *attempts* to correct errors. NAFP sends a single NACK for each missing fragment, expecting the sender to receive the NACK and retransmit the missing fragment. If the simple, low cost, NAFP error correction mechanism fails, the XPP layer will eventually detect the lost message and take the corrective measures needed to reliably deliver the message. Because we assume the communication channel delivers most packets without error, the NAFP protocol optimizes for the expected case where no errors occur by not acknowledging individual fragments. As long as no errors occur, NAFP transfers and reassembles XPP messages using the minimum number of packets with minimal computational overhead. When a fragmentation error does arise, NAFP attempts to correct the error using an inexpensive NACK message. In practice, a single NACK message corrects most errors. If the original fragment was delayed and not lost, the receiver will receive a duplicate fragment. In this case, NAFP simply discards the duplicate fragment.

5.4.3 Message Format

Figure 5.7 shows the format of an NAFP message. Each NAFP packet consists of a header area and a data area. The header area contains a *HEADER LEN* field that specifies the length of the NAFP header. The header length depends on the options used in the *OPTIONS* field of the header. NAFP uses the *OPTIONS* field to provide options such as checksums.

NAFP supports two types of messages: messages containing XPP fragments and messages carrying NACKs. The *TYPE* field in the header specifies the NAFP message type. NAFP identifies all the fragments that belong to an XPP message with a

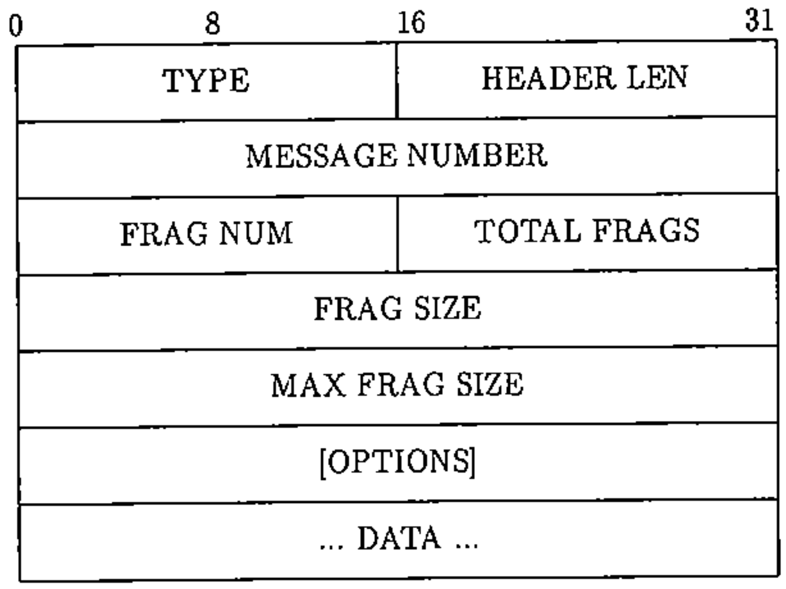

Figure 5.7 The format of an NAFP packet.

MESSAGE NUMBER unique to each client. The *TOTAL FRAGS* field specifies the number of fragments that make up the XPP message, while the *FRAG NUM* field specifies which fragment of the sequence the current packet contains. The *FRAG SIZE* specifies the size of the data area. For all but the last fragment, *FRAG SIZE* equals the *MAX FRAG SIZE*. NAFP uses the *MAX FRAG SIZE* to determine the position of the current fragment in the XPP message and deposits the fragment directly into the buffer for reassembly.

In our implementation, the NAFP protocol only sends NACKs from the server to the client. Clients never send NACKs to the server. There are two reasons why we chose to implement the protocol in this way. First, the XPP protocol simplifies the implementation of NACKs in the case of messages flowing from the client to the server but does not simplify the implementation of NACKs for messages flowing from the server to the client. When a client sends an XPP message to the server, the XPP layer saves the message until it receives an XPP reply message from the server. Because the XPP layer saves the message, the NAFP layer has the message available in case it receives a NACK and needs to resend one of the fragments of the message. However, when the memory server sends an XPP reply message, the XPP layer deletes the reply message as soon as the NAFP layer sends it. To implement NACKs for messages traveling from the server to the client, the NAFP layer would have to save a copy of the message, possibly wasting valuable buffer space on the server. In addition, clients do not acknowledge receipt of XPP reply messages, making it difficult for the NAFP layer on the server to know how long to save the message.

The second reason for not implementing NACKs for messages flowing from the server to the client arises from a desire to make the server as efficient as possible. In the remote memory communication protocol, reliability is the responsibility of the client. As a result, the server is substantially simpler and more efficient than a server that must provide reliability. Because NAFP does not use NACKs to correct messages traveling from the server to the clients, the server does not need to save messages, time-out messages, handle incoming NACK packets, or retransmit fragments.

Simplifying the protocol at the server allows the server to spend more time processing requests and less time transmitting them. Moreover, because the server handles multiple requests from multiple clients simultaneously, the server is more likely than the client to drop a packet due to a queue overflow. In practice the error rate for messages flowing from the clients to the server tends to be higher than the error rate for messages flowing from the server to the clients.

5.4.4 NAFP as a General Technique

The NAFP protocol serves as an efficient transport protocol for the XPP protocol. However, we can use the techniques used by NAFP to improve the performance of almost any high-level protocol that provides reliable delivery over a datagram-based communication channel.

For example, the TCP [Com88] protocol provides reliable delivery over physical network architectures that provide datagram service. The IP layer provides fragmentation for TCP, but the TCP layer provides the reliable delivery. If the IP layer loses a fragment from a TCP message, the TCP layer will not detect the error until the message times-out. TCP then resends the entire message. Placing the NAFP layer between the TCP layer and the IP layer would give NAFP a chance to correct fragmentation errors before they reach the TCP layer. NAFP techniques could enhance the performance of other reliable protocols such as TP4, VMTP, TFTP, NETBLT, and RPC as well [Org86, Che88, Sol81, CLZ87, Lyo84].

5.5 Related Work

Before designing the Remote Memory Communication Protocol, we examined several protocols as possible remote memory paging protocols. Unfortunately, existing protocols do not provide the functionality or performance we desired.

Hardware protocols severely limit the portability of the system. In addition, protocols like the Ethernet protocol or the proNET protocol do not provide reliable delivery [Dig80]. Other protocols like Digital's DDCMP protocol [Dig84] provide reliable datagram delivery but do not provide fragmentation.

Existing software protocols do not offer the desired functionality or do so at the expense of performance. The UDP protocol [Pos80b], for example, allows arbitrarily large datagram messages up to 64K bytes, but does not provide reliability. The VMTP protocol [Che86, Che88] does provide reliable delivery of arbitrarily large datagrams; however, it uses positive, selective acknowledgements that increase the number of packets transmitted and reduce the efficiency. The TCP protocol [Pos81, Com88], like VMTP, uses positive ACKs to provide reliable delivery, increasing the number of messages transmitted. In addition, the high cost of setting up and tearing down a TCP connection reduces the efficiency of the protocol, especially when the application is not connection oriented. Moreover, because paging lends itself more to the datagram abstraction than to the byte stream abstraction provided by TCP, paging across TCP would require simulation of datagrams using streams and record marking.

Sun's NFS/RPC [SGK$^+$85, San85, Lyo84] protocols provide functionality similar to RMCP. Both protocols only use two messages to access a server in the common case. However, Sun NFS/RPC does not support data streaming and depends on the UDP protocol for data transfer. Sun NFS/RPC does not handle fragmentation, queueing, multiplexing, or demultiplexing, but instead relies on UDP to provide this functionality. The dependency on UDP prohibits use of NFS/RPC over virtual or physical transport protocols that do not provide fragmentation and reassembly.

RMCP does its own fragmentation and queueing, and runs over any protocol that provides datagram delivery. Consequently, paging performance can be improved by building the RMCP protocol directly on top of the physical network architecture as opposed to a high-level virtual network architecture like IP (e.g., When all clients and hosts reside on a single Ethernet, RMCP can use the Ethernet link-level protocol directly).

In addition, the UDP(IP) protocol used by NFS/RPC makes no attempt to retransmit lost fragments [Pos80a]. If IP loses a fragment, it discards the message, causing NFS/RPC to timeout and resend the entire message. Fragment loss increases when messages travel across gateways or when a fast host sends to a slow host. Consequently, the NFS/RPC protocol sets an upper limit on the size of NFS messages to reduce the number of fragments per message. The RMCP protocol does not limit the size of a message, transferring arbitrarily large pages without degrading performance.

The Sprite RPC protocol [Wel86], like Xinu's RMCP protocol, allows arbitrarily large messages and performs fragmentation and reassembly of messages. However, Sprite RPC does not provide data streaming. Sprite RPC will not send a new RPC message until the sender receives a reply for the previous RPC message. The protocol only allows synchronous messages because it uses request messages as acknowledgements for reply messages. The lack of asynchronous message delivery prohibits concurrent message processing and reduces throughput.

Sprite's method for handling fragmentation differs substantially from RMCP. Sprite RPC uses partial acknowledgements to correct fragmentation errors. Although partial acknowledgements correct fragmentation errors, they add considerable complexity and overhead to the protocol and create implementation problems [Wel86]. Because the receiver does not know whether a missing fragment was lost or delayed, deciding when to send a partial acknowledgement is difficult. Sprite RPC sends a partial acknowledgement when it receives the last fragment. If the last fragment is lost, the protocol uses a complex set of rules to determine when to send the partial acknowledgement. RMCP does not suffer from the lost last fragment problem because it sends a NACK as soon as it detects a missing fragment. Moreover, because RMCP uses data streaming, it detects missing last fragments when it receives the first fragment of the next message. Another disadvantage of Sprite RPC is the added overhead incurred by repeatedly transmitting missing fragments. Sprite attempts to reliably deliver the fragments of a message and may perform several iterations of partial-acks, time-outs, and retransmissions.

5.6 Summary

In this chapter we describe the design of a highly-efficient, reliable, architecture-independent remote memory communication protocol. Each client's virtual memory system uses the protocol to reliably access remote memory backing storage.

The Remote Memory Communication Protocol consists of two layers: the *Xinu Paging Protocol* (XPP) layer and the *Negative Acknowledgement Fragmentation Protocol* (NAFP) layer. Dividing the protocol into two distinct layers clearly separates the high level paging operations from the low level details of data transfer across the communication channel.

The XPP protocol uses timeouts, retransmissions, and positive acknowledgements to guarantee reliable access to the remote memory backing store. XPP assigns sequence numbers to every XPP message to guarantee in-order processing of page requests. To improve efficiency without affecting paging semantics, XPP uses a preceding message number to permit delivery of messages that arrive out-of-order but do not violate the partial ordering on the list of messages. To increase throughput, XPP provides data streaming, allowing the virtual memory system to issue multiple, concurrent, paging requests.

The NAFP protocol implements the low level transport operations required to transfer XPP messages across the communication channel. NAFP hides the characteristics of the communication channel and the details of message delivery from the XPP layer, allowing the XPP layer to remain independent from the underlying network architecture. In particular, NAFP fragments and reassembles XPP messages, allowing the size of an XPP message to exceed the MTU of the communication channel. To improve efficiency, NAFP uses negative acknowledgements (NACKs) to correct fragmentation errors as soon as they occur. The result is a highly-efficient, portable, paging protocol.

6. A REMOTE MEMORY BACKING STORE

The Remote Memory Model centers around the use of remote memory servers for backing storage. The functionality and performance provided by a memory server separates the Remote Memory Model from conventional distributed systems. As you will recall from chapter 3, the remote memory model consists of one or more remote memory servers that provide shared, high-speed remote memory storage to heterogeneous client machines. Each memory server has a large physical memory which is shared by all the client machines. In addition, each memory server has additional secondary storage space available in the event that clients collectively exhaust the server's large physical memory.

This chapter focuses on the design of the remote memory server. In particular, we outline our goals for the memory server and show how the design obtains these goals. Although the server can be viewed as a general purpose, high-speed remote memory storage device, the server's design is ideally suited for client virtual memory systems requiring remote memory backing store support.

6.1 Design Goals

Our objective was to design a memory server capable of providing remote memory backing storage to multiple client machines simultaneously. More specifically, the design should achieve the following goals:

Heterogeneous Client Support: The server should provide remote memory backing storage to multiple heterogeneous client machines simultaneously.

Resource Sharing: Instead of preallocating fixed amounts of memory to each client, client machines should share the memory resource based on their needs.

High Level Abstraction: The server should provide a high level abstraction that hides the details of data storage from the clients.

Arbitrarily Large Storage: The server should provide clients with an arbitrarily large storage space.

Scalability: Server performance should not decrease as the number of clients increases.

Fast Data Access: To improve overall efficiency, the server must efficiently locate and retrieve stored data.

The remote memory model assumes the client machines consist of a variety of hardware architectures (e.g., pages size, word size, byte order) and operating systems. Consequently, a remote memory server must expect and support heterogeneous clients. The memory server should provide the functionality required to support a wide variety of client virtual memory systems. However, to achieve high-speed access times, the memory server should choose simple, efficient mechanisms over complex, time-consuming mechanisms whenever possible. The design must choose an appropriate balance between functionality and efficiency/simplicity.

To maximize sharing of the remote memory resource, the memory server should avoid preallocating fixed amounts of its physical memory to each client machine. Instead, the memory server should dynamically allocate its memory space to clients based on their current needs.

Conventional systems that use remote backing storage (e.g., diskless systems paging to a remote file system) provide a low level interface to unstructured files or disks [SGK+85, OCD+87, TvRvS+90]. In these systems, the server places the responsibility of managing the backing store resource (i.e., allocation, deallocation) on the client's virtual memory system. The client's virtual memory system keeps information about the regions of the backing store currently in use and maintains a *free list* identifying the unused regions of the disk.

Instead of requiring the clients to manage the backing store (or worse yet, requiring the clients to cooperatively manage the backing store), the memory server should present client machines with a high-level abstract service. In particular, the abstraction should hide the underlying storage mechanism and storage structure from the client machines, thereby freeing clients from the responsibility of managing the storage space. Hiding the internal structure and storage mechanism from the clients allows the server to efficiently manage the physical memory storage space and equitably allocate memory resources to the clients based on their needs. Hiding allocation of the server's physical memory resources frees clients from the responsibility of cooperatively managing the backing store.

In addition, the abstraction should hide the server's physical memory size from the clients. The server's physical memory size should not constrain the amount of remote storage space obtainable by a client. Just as virtual memory operating systems provide programmers with large virtual memory spaces that do not depend on the target machine's physical memory size, a memory server should present its clients with an arbitrarily large storage space. A client machine should be able to select and use any memory server for backing storage, regardless of the server's physical memory size.

The performance of a client's virtual memory system depends heavily on the remote memory server's performance. When a process executing on a client machine attempts to access a page that has been stored on a remote memory server, it blocks until the virtual memory system has retrieved the missing page from the memory server. Reducing the delay associated with retrieving data from the remote memory backing store reduces the time the process remains blocked and results in improved client performance.

The remote memory access time can be divided into two components: *protocol overhead* and *server overhead*. Chapter 5 describes an efficient communication protocol with low overhead. The second component, server overhead, consists of the time to decipher the request, verify the validity of the request, determine the location of the requested data in the server's memory, and store or fetch the data to or from the server's memory. Moreover, because the server hides the internal storage structure from the clients, the server overhead includes the time spent managing the storage space. The design should minimize the time spent performing these operations, thereby reducing the server overhead. Reducing the server overhead increases the number of requests per second processed by the server. An increase in the number of requests per second handled by the server means an increase in the number of client machines a server can efficiently support.[1]

6.2 A Logical Memory Server

To hide the internal structure and organization from the clients, each memory server provides a high-level abstraction called a *Logical Memory Server* (LMS) which clients use as the interface to the memory server. Each memory server machine provides multiple Logical Memory Servers and assigns exactly one LMS to each client machine requiring backing storage. Figure 6.1 illustrates a memory server machine providing multiple LMSs to multiple client machines. When a client's virtual memory system requests access to the memory server, the server creates a new LMS for the client. The memory server only allows client machines to access their own LMS, thereby protecting each client's LMS from all other clients. The server uses highly-efficient, but flexible, data structures and algorithms to implement the abstraction. The flexibility of the abstraction allows heterogeneous client machines to simultaneously access the same server.

A *Logical Memory Server* consists of a structured memory space and a set of operations on the memory space. Instead of providing an unstructured array of memory cells and requiring the clients to manage the space, the LMS imposes a structure on the memory space. Figure 6.2 illustrates the organization of the memory space provided by an LMS. As the name implies, each Logical Memory Server provides the client it serves with a large virtual (logical) memory space. An LMS organizes the space into *Virtual Segments* (VS), each subdivided into *pages*. The smallest unit of storage on an LMS is a single page. However, the abstraction does not specify

[1]Alternatively, the server can support a larger workload from the same number of clients.

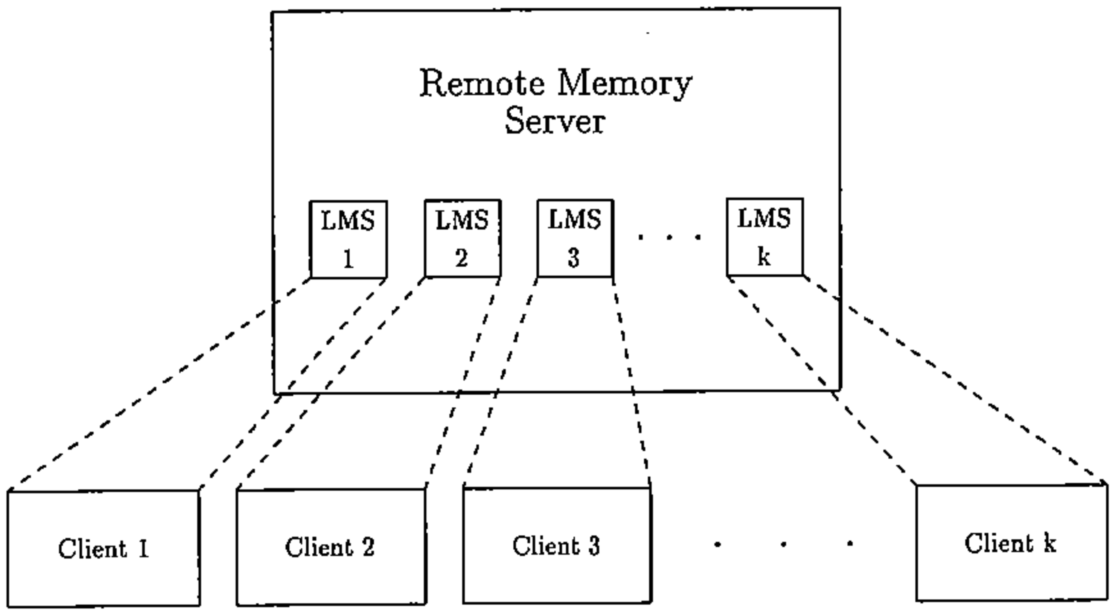

Figure 6.1 The remote memory server provides each client with its own *Logical Memory Server* (LMS) and regulates/protects the LMS from access by other clients.

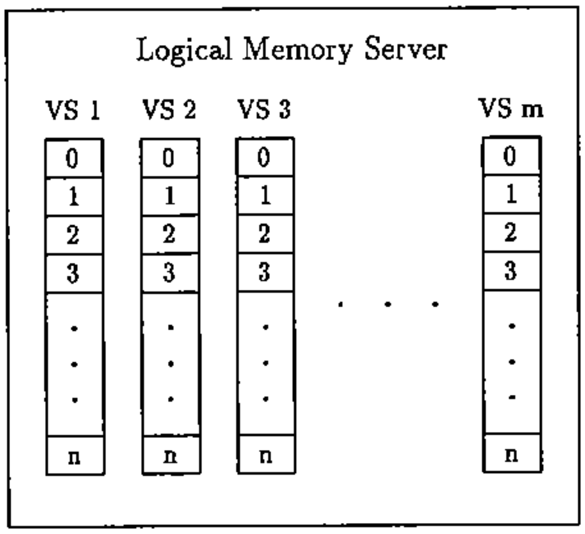

Figure 6.2 The organization of a *Logical Memory Server*. A Logical Memory Server organizes the memory space into *Virtual Segments* (VS), each subdivided into *pages*.

the size of a page. LMS pages are virtual pages, and any two LMSs may have different page sizes. Because the memory server creates a new LMS for each client machine, the memory server allows the client to define the size of the pages in its LMS. When a client connects to a memory server, the client sets the page size for its LMS. Consequently, the abstraction allows the memory server to adapt to meet the needs of the heterogeneous client architectures that access the server.

An LMS provides clients with four abstract operations that manipulate the memory space. Clients can *activate* or *deactivate* a Virtual Segment in their LMS and *read* or *write* individual pages of a Virtual Segment. Before a client can store data on the memory server, it must first activate the VS into which it wants to store the data (assuming the VS is not already active). After the VS has been activated, the client can write data into the pages of the VS. Similarly, a client may read individual pages from a VS or deactivate a VS when it is no longer needed.

Virtual Segments in an LMS are numbered from 0 to m, where m is the maximum number of Virtual Segments per LMS. Similarly, the memory server numbers the pages within a VS from 0 to n, where n is a limit on the number of pages per

VS. The abstraction does not specify a limit on the number of Virtual Segments or the number of pages per Virtual Segment. However, every memory server imposes an implementation-dependent limit on m and n much like computer architectures define an architecture-dependent limit on the size of a virtual address. Clients use the VS id and page number to uniquely identify a page in an LMS. When activating/deactivating a VS, the client specifies the VS id as part of the activate/deactivate request. To read or write a page in a VS, the client specifies a (VS id, page number) pair.

The memory server presents the same abstraction to all the client machines it serves. The client's virtual memory system must map the client's address space layout and virtual memory structures onto the abstraction provided by the server. In practice, the functionality provided by the abstraction mirrors the operations required by the client's virtual memory system, resulting in a trivial mapping function. For example, many conventional virtual memory operating systems support separate address spaces for each process. Consequently, the client's virtual memory system uses a straight-forward, one-to-one mapping between processes on the client and Virtual Segments on the server. Even operating systems with more elaborate or complex virtual memory organizations map nicely onto the abstraction. For example, VM Xinu supports multi-threaded user-level processes. Threads within a process share portions of the address space that contain global data, but maintain per-thread mappings for the regions of the address space that contain private data. Xinu's virtual memory system activates one VS on the LMS for each thread's private data and an additional VS for the data shared by all the threads. Despite the added complexity, the abstraction facilitates a trivial mapping.

The abstraction provides a simple, yet powerful, remote storage service. Despite its simplicity, the functionality provided by the abstraction often eliminates the need for clients to manage the storage space on the server. Clients do not have to deal with allocating or freeing pages on the server, maintaining a list of free pages, or remembering the location of data stored on the server, because the structure of an LMS mirrors the virtual memory organization on the client.

The abstraction hides the internal structure and implementation from the clients. This allows the implementer to choose highly-efficient data structures and algorithms tailored for the architecture on which the server executes. In addition, the abstraction hides the architecture-dependent characteristics of the memory server, such as the server's physical memory size. Consequently, a client may use any memory server, regardless of the amount of memory physically present on the server.

The abstraction also allows the memory server to dynamically allocate physical memory to clients based on their needs. The memory server only maps as much physical memory into a client's LMS as the client needs. As a result, all the clients share the memory server's physical memory resource. Moreover, the flexibility of the abstraction to adjust an LMS's page size based on the client's page size allows the memory server to support clients with a wide variety of page sizes simultaneously.

Although the abstraction described here does not provide operations for sharing data between clients, the abstraction does not prohibit such operations from being added in the future. Because an LMS provides virtual pages, the abstraction could be extended to allow clients to map two or more virtual pages from different Logical Memory Servers to the same data, thereby allowing clients to share data.

6.3 The Design of a Logical Memory Server

Because the abstraction completely defines the client interface to remote memory backing storage, client machines only interact with the abstraction and never see the internal structure and organization of the remote memory server. Consequently, the remote memory server may use any data structures or algorithms it desires to efficiently implement the abstraction.

6.3.1 Managing the Virtual Segments

Several factors influence the speed at which a memory server can process requests. As the number of clients increases, the number of Logical Memory Servers supported by the memory server increases. In addition, an increase in the number of clients per memory server typically results in an increase in the amount of data stored on, and managed by, the server. Increasing the number of clients or the amount of data stored on the server can cause a significant increase in the server's overhead, resulting in degraded server performance. For many conventional servers there exists a direct correlation between the load on the server (e.g., number of clients) and the server's response time [LS89, LZCZ86]. As the server load rises, response times increase and client/server performance decreases. Our goal was to design a remote memory server that would not decrease in performance as the number of clients or the amount of data stored on the server increases.

The memory server must maintain mappings for the pages in all the Virtual Segments in each LMS. As the number of clients increases, the number of LMSs, VSs, and pages managed by the server increases. That is, supporting the abstraction poses a potential bottleneck and limitation on the scalability of the model. The overhead required to support the abstraction could significantly increase the time required to deposit or retrieve data to or from the server.

To reduce the delay associated with retrieving memory from the memory server, the server must use highly-efficient algorithms and data structures. The memory server cannot waste time searching data structures for the desired data. Moreover, to provide as much physical memory as possible for client data, the server should reduce the amount of memory used by the data structures that manage the storage space. Clearly, the server cannot afford to maintain a mapping for every page.

In our prototype, the server stores page mappings in a hash table, called the *virtual-page hash table*. The virtual-page hash table only stores pages that contain data. Unused pages do not appear in the table. Because the only operations allowed on a page are store and fetch operations, the server efficiently processes paging requests using the insert and look-up operations of the hash table. Each hash table entry contains the following information:

Logical Memory Server ID: The LMS identifier differentiates between the various Logical Memory Servers active on the memory server.

Virtual Segment ID: The VS identifier specifies the Virtual Segment to which the data belongs.

Page Number: The page number identifies a specific page within a VS.

Data Pointer(s): The data pointer indicates the physical memory location at which the server can find the page's data.

Timestamp: The timestamp field records the timestamp of the VS to which the data belongs.

Client machines uniquely label each page with an ordered triple containing the LMS ID, VS ID, and page number. Given a paging request, the server can quickly verify whether a particular hash table entry contains the requested page.

Because each hash table entry contains all the information required to uniquely identify a page, the server stores all pages, regardless of the VS or LMS to which a page belongs, in the virtual-page hash table. Using a single hash table to store all pages significantly simplifies the server's design. The server does not waste time (or code) managing multiple (similar) data structures (e.g., allocation/reclamation of entries) and does not need to limit the number of clients it serves based on a predefined, maximum number of hash tables.

Hash table entries do not contain any data. Instead, each hash table entry contains a pointer to the data. The indirection allows flexible placement of data, arbitrary size pages, and does not consume valuable memory space for unused hash table entries. Consequently, the hash table can be arbitrarily large without consuming an exorbitant amount of physical memory. The indirection also allows the server to store the data on secondary storage when desired.

The memory server efficiently locates a hash table entry by applying a double hashing algorithm to the ordered triple that uniquely identifies the desired page. For example, assume the memory server receives a fetch request. The server extracts the LMS ID, VS ID, and page number from the request to form an ordered triple that uniquely identifies the requested page. While probing the hash table, the double hashing algorithm compares the triple constructed from the request to the triple stored in the hash table. When a match is found, the server fills in the reply message with the data pointed to by the hash table entry. In the case of a store request, the server locates an unused hash table entry and initializes the entry to point to the data in the message.

Double hashing has the desirable property of locating the desired data, or an entry in which to store the data, in constant time on average [Knu73]. If the hash table is less than 95% full, the average number of probes used by the double hashing algorithm to locate the specified data does not exceed three. As long as the remote memory server limits the utilization of the hash table to 95% of its total capacity, the average look-up time remains constant, regardless of the number of entries used (i.e., the number of pages stored on the server). The single hash table, together with the double hashing algorithm, allow the memory server to serve multiple clients without a degradation in performance. Consequently, the server scales well as the number of clients increases.

Although we assume the memory server has a large physical memory, there still exists an architecture-imposed limit on the server's physical memory size. Memory consumed by the hash table reduces the amount of memory available for client data. Because the server uses a single hash table to map all data stored on the server, the hash table must have enough entries to map all the storage space available on the server. However, to increase the amount of physical memory available for data, the server starts with a hash table only large enough to map the physical memory storage space, not secondary storage. When the server exceeds 95% of the hash table's capacity, it dynamically increases the size of the hash table to insure the average access time remains constant. The ability to dynamically grow the hash table allows the server to maximize the amount of physical memory available for storage and still provide efficient remote memory access.

6.3.2 Physical Memory Management

The memory server must make efficient use of the physical memory in order to increase the amount of physical memory available for client data. As stated earlier, to increase the amount of physical memory available for client data, virtual-page hash table entries do not reserve memory space for client data. Instead, each hash table entry stores pointers to the memory where the data can be found. This design decision allows the server to choose a physical memory allocation strategy that makes the most efficient use of the available memory.

The memory server divides the available physical memory into small fixed-size segments called *data blocks*. The memory server dynamically allocates and assigns data blocks to each new store request. If the size of the page in the store request exceeds the size of a data block, the server allocates multiple data blocks for the page.

The tradeoff between data block overhead and memory space utilization makes it difficult to choose an optimal data block size. Using large data blocks causes internal memory fragmentation, while small data blocks increase management overhead. In theory, the server defines the data block size as the smallest common denominator of all the client page sizes such that the overhead is still tolerable. In practice, only a few popular page sizes exist based on powers of two, making the choice easy.

The hash table does not require the server to store the data in contiguous data blocks. If the server cannot find a set of contiguous data blocks large enough to store the page, the server spreads the page across several non-contiguous data blocks. The ability to scatter the data from a store request across multiple non-contiguous data blocks results in efficient use of memory and support for a wide variety of page sizes. In addition, the server can often allocate contiguous data blocks to a page and save only the starting address in the hash table. If the server must spread the page across multiple non-contiguous data blocks, it dynamically allocates an array of pointers to the data and fills in the hash table entry to point to the array of pointers. The indirection reduces the size of the hash table and provides all the necessary information for the common case; the case when contiguous data blocks are available.

6.3.3 A Transparent Two-level Storage Space

Because each memory server machine has a fixed amount of physical memory (either due to hardware constraints or economic constraints), the LMS allows the implementation to use secondary storage to store additional client data. The ability to use secondary storage allows the server to provide an arbitrarily large amount of storage space. The LMS hides the location of the data from the clients, allowing the implementation to store the data on secondary storage as well as in physical memory. Possible secondary storage media include local disks, remote disks, second-level memory (e.g., a global [vs. local] memory on a multiprocessor), remote memory, or some combination of the above.

The server manages the secondary storage by dividing the secondary storage space into fixed-size data blocks equal in size to the physical memory data blocks. Each hash table entry maintains a list of pointers to the physical memory and secondary storage data blocks that make up a page.

The server manages its physical memory and secondary storage much like a virtual memory operating system. A replacement policy determines which pages may remain in memory and which should be moved to secondary storage. To reduce the delay associated with accessing remote memory, the server chooses a replacement policy that best fits the access patterns of the clients it serves. The replacement policy also affects the level of resource sharing allowed by the server. If the server wants to limit the amount of physical memory obtainable by a client machine, the server may employ a local replacement algorithm, allocating a fixed amount of memory to each client. As stated in section 6.1, our goal was to share the memory resource fairly among all clients based on their needs. Consequently, the server uses a global replacement policy which does not limit the amount of memory attainable by a client or process.

The server may use two or more different types of secondary storage to increase its storage capacity. In this case, the replacement policy must manage a multi-leveled memory. For example, assume the memory server executes on a multiprocessor NUMA architecture[2] in which each processor has access to high-speed local memory, global memory (slower than the local memory), and static disk storage. In this case, the replacement policy would determine data placement based on the relative speeds of the various levels of the memory. The ability to select a replacement policy based on the architecture of the memory server or the access patterns of clients allows the server to execute on a wide variety of architectures and secondary storage configurations. The abstraction provided by the server hides the two-level storage space and the location of the data from the clients. From the client's viewpoint, the server simply provides a single, large memory resource.

6.3.4 Memory Reclamation

Logical Memory Servers do not explicitly provide an operation to release individual pages of client data. Instead, the only means for a client to release memory held on the server is through a *terminate* request or a *shutdown* request. A terminate request indicates that the client's virtual memory system no longer needs the data stored in the specified VS. The client's virtual memory system typically issues a terminate request when a process on the client terminates. Similarly, a shutdown request indicates that a client is shutting down and no longer needs the data it stored on the server.

There are several reasons that influenced the design decision to use terminate and shutdown requests instead of release page requests. First, many conventional virtual

[2]NUMA is an acronym for Non-Uniform Memory Access [Tev87, BFS89].

memory systems, and also the VM Xinu system, only release backing storage space when a process terminates [Bac86, CG91, LL82, Ras86]. When the virtual memory system retrieves data from the backing store, it does not remove the data from the backing store. For example, imagine an application attempts to access data on page N, which has been written out to disk. The virtual memory system reads in the contents of page N from the disk but does not erase the copy of page N stored on the disk. If the application does not modify the contents of page N, the virtual memory system can discard the data and reclaim the memory because a copy of the data still exists on the disk. Second, when a process terminates, the client does not need to traverse its page tables looking for pages on the backing store to release. Instead, the server remembers all the pages associated with a process and frees them for the client. Third, allowing clients to release multiple pages in a single operation, as opposed to releasing individual pages, reduces the amount of network traffic generated when a process terminates. It also reduces the number of requests the server must process.

When a process terminates on a client machine, the client issues a terminate request to release the memory on the server owned by the process. The terminated process may have amassed a significant amount of storage space which the server must locate and return to the free list. Consequently, the time required to reclaim the memory may be substantial and is directly proportional to the size of the process. However, the memory server must continue to process incoming requests and cannot afford to devote a large amount of processing time to any single request. In particular, terminate requests should not force the server to spend a long period of time reclaiming memory. In addition, the memory server should not delay the client by delaying the terminate reply. Ideally, a memory server should process terminate requests in constant time, regardless of the size of the terminated process.

The memory server processes terminate requests in constant time, regardless of the size of the terminated process. Each time the memory server receives a terminate request, it records the request and immediately sends a terminate reply message to the client. The server remembers the request and delays the memory reclamation until later. Similarly, upon receiving a shutdown request, the server immediately sends a shutdown reply, and delays reclamation of the memory until some time in the future.

The memory server uses a novel technique to amortize the cost of memory reclamation over time. Section 6.3.1 described how the *virtual-page hash table* allows the server to locate data in constant time. However, the hash table data structure does not lend itself to efficient page reclamation. To locate all the pages in a particular VS, the memory server must sequentially traverse the entire hash table.[3] To reduce the cost of reclaiming memory, the memory server uses *timestamps* and two additional hash tables: the *LMS hash table* and the *VS hash table* (see Figure 6.3).

The LMS hash table maintains information about the active Logical Memory Servers. The VS hash table maintains information about all the active Virtual Segments. Each VS hash table entry contains a pointer to the parent entry in the LMS table to which the VS belongs. Similarly, every virtual-page hash table entry contains a pointer to the VS table entry to which the page belongs. Storing pointers to parent tables allows the server to find all the information about a particular page in constant time. In addition, pointers reduce the space consumed by the virtual-page hash table by only storing a single copy of information that is shared by many pages.

The memory server uses *timestamps* to identify expired data belonging to terminated processes or clients. The memory server assigns a timestamp to every LMS and to every VS within an LMS. The server saves the *current timestamp* of every LMS in the LMS hash table and saves the timestamp of every VS in the VS hash table. Each virtual-page hash table entry records the timestamp of the VS to which the data belongs. When the memory server receives a page store request, the server saves the timestamp of the requesting process in the virtual-page hash table entry. Consequently, all pages belonging to a VS have the VS's timestamp. When the memory server receives a terminate request, it updates the current timestamp in the VS table, thereby invalidating all the pages in the VS. Similarly, when the memory server

[3]Alternatively, the server could sequentially traverse the entire VS.

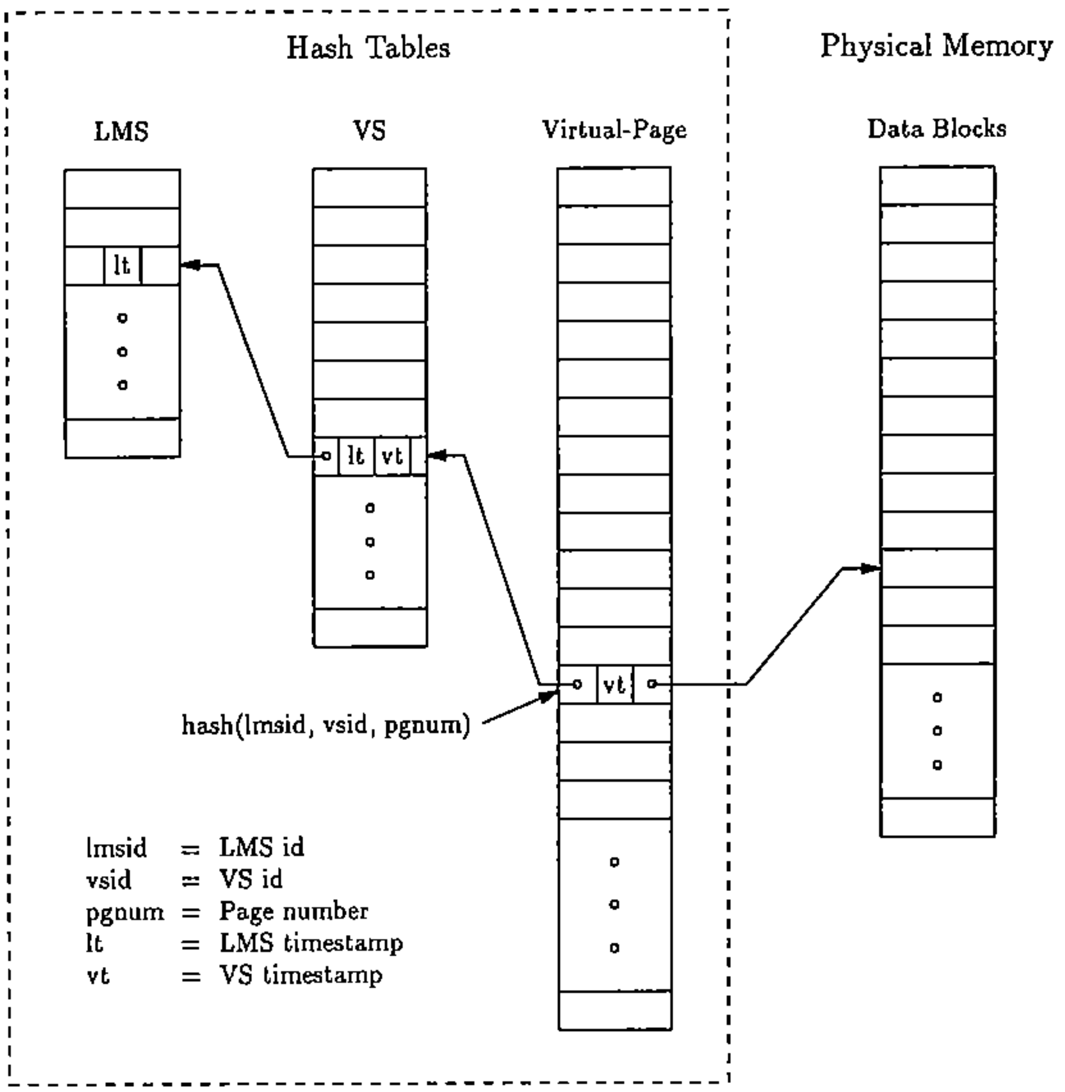

Figure 6.3 The three hash tables: LMS hash table, VS hash table, and the virtual-page hash table. Each virtual-page table entry contains the owner's timestamp and a pointer to the owner's VS table entry. Each VS table entry contains the current timestamp for the VS, an LMS timestamp, and a pointer to the owner's LMS table entry.

receives a shutdown request, the server updates the current timestamp in the LMS table which invalidates all the data associated with the LMS. In addition, timestamps provide a convenient mechanism for handling client failure. When a client crashes and reboots, the memory server assigns a new timestamp to the client and automatically reclaims the memory space used during the client's previous lifetime.

Timestamps allow the server to process terminate and shutdown requests in constant time. Updating a timestamp invalidates all the data in a VS or LMS in a single operation. Timestamps allow the server to immediately send a terminate or shutdown reply message and postpone memory reclamation until later.

The memory server uses two methods to reclaim invalid pages. The first method reclaims memory while processing store and fetch requests. When a client issues a store or fetch request, the server applies the double hashing algorithm to locate an entry in the virtual-page hash table. The double hashing algorithm requires the server to check for a collision on each probe to the table. That is, the server must compare the requested page's information against the information found in the hash table entry to see if the entry contains the desired page. If a collision occurs, the server checks the timestamp found in the hash table entry against the owner's timestamp in the VS and LMS hash tables. If the timestamps differ, the server reclaims the page. If the timestamps match, the hash table entry is still valid, and the double hashing algorithm proceeds as normal. This modification to the double hashing algorithm allows the server to reclaim invalid pages during its normal processing.

The second and more conventional method makes use of a garbage collection thread within the memory server. A garbage collection thread executes in the background at a lower priority than the memory server so that it only executes when the server is not servicing client requests. It traverses the virtual-page hash table in search of pages with invalid timestamps and returns them to the free list.

Because the memory server handles all memory reclamation, clients do not have to remember and free individual pages stored on the server. Instead, the server allows clients to free all the pages associated with a terminated process in a single operation. Timestamps allow the memory server to process terminate requests and shutdown requests in constant time by delaying memory reclamation. The garbage collection thread together with the lazy reclamation modification to the double hashing algorithm amortize the cost of reclaiming memory over time.

6.4 Enhancing Performance

The flexibility of the memory server's design allows the implementation to take advantage of emerging multiprocessor architectures and high-bandwidth networks to enhance performance.

First, the design lends itself nicely to a multiprocessor implementation. The server receives independent paging requests from multiple clients simultaneously. Because these requests are independent of one another, a multi-threaded server executing on multiple processors could significantly improve the client's performance. In addition, the design allows the server to process requests from an individual client machine out-of-order. Consequently, a multiprocessor could improve an individual client's performance by processing its requests in parallel. Moreover, the background processing performed by the garbage collection thread maps nicely onto a multiprocessor system.

Second, the transparent two-level backing store allows memory servers to use other memory servers for backing storage. High-bandwidth, low-delay, local and metropolitan area networks make remote memory a desirable form of secondary storage for memory servers as well as for client machines. Moreover, the two-level backing store allows memory servers to be chained together linearly or hierarchically to provide an arbitrarily large amount of remote memory backing storage.

6.5 Related Work

Several distributed systems use a remote backing store to support diskless nodes [GMS88, LLD+83, PPTT90]. Instead of using a local disk or file system for backing storage, these systems provide remote file system support in the kernel and allow the virtual memory system to access the remote file system as if it were accessing a local file system. The advantage of providing a transparent remote file system is that the virtual memory system requires no modifications and is completely unaware of the location of the files used for backing storage.

Early versions of the SunOS operating system [GMS88] maintained Unix's notion of a fixed-size, preallocated, swap partition on the local disk, but modified the lower level drivers to allow access to a remote disk as opposed to a local disk. The server assigned a *network disk* partition to each client and allowed the client to read and write disk blocks from the partition as if it were a local disk. More recent versions of the SunOS operating system allow clients to use remote files for backing storage. The abstraction of files hides the underlying storage device and its location from the virtual memory system and allows the operating system to use high-level file operations to access the backing store. Despite this added level of abstraction, the client's virtual memory system must still perform the bookkeeping required to manage the backing store (swap file), much like earlier versions managed disk partitions. In addition, the server preallocates a fixed-size, private, swap file for each client. Preallocation consumes valuable disk space that could otherwise be shared among the clients. Each client is limited by the disk space in its own swap file and cannot access unused disk space in other client's swap files. Because clients independently manage their swap files, no data sharing between clients is allowed. Consequently, swap files often contain identical data which cannot be shared.

The model in which file servers double as the backing store has several disadvantages. First, the model simply extends the conventional disk backing store to a distributed system with remote disks or files. Virtual memory systems that use file servers for additional memory space must pay the additional overhead incurred by the file system and the cost of committing the data to disk. The enormous performance difference between remote memory and remote disk significantly impacts the performance of the virtual memory system. Second, the file server cannot distinguish between file data and paging data. Consequently, the server cannot make intelligent decisions regarding caching. File servers optimize for the most common file access method: sequential file access. Paging activity tends to randomly access pages on the backing store, rendering the file server's optimizations useless. Optimizing caching for the sequential case wastes valuable buffer space and degrades both paging performance and file system performance. Third, the client virtual memory system competes with ordinary user-level processes for the file server's resources. They compete for the server's CPU time, the server's buffer cache, and disk space. The virtual memory system has no special privileges. As a result, the paging system cannot guarantee high-speed access to the backing store.

Other example operating systems that use the file server model (in addition to SunOS) include the Sprite operating system [OCD+87] and the Amoeba distributed operating system [TvRvS+90, RST89]. These systems also suffer from many of the remote file system problems described earlier. In fact, the optimizations used in the Amoeba file server are even more detrimental to paging performance. Whenever a client accesses a remote file, the Amoeba file server, called the *Bullet File Server*, prefetches the entire file assuming the client will access the entire file in the near future; an extreme form of Read-Ahead. In addition, this form of prefetching limits the size of a file to relatively small files that fit in the memory of the file server. The Sprite system overcomes some of the problems found in SunOS by using multiple files, as opposed to a single file, for backing storage. In addition, Sprite does not preallocate fixed-size swap files, thereby allowing clients to share the resource.

Another related form of backing storage are *caching disks*.[4] Although the idea is not necessarily new [Gro89], caching disks are becoming more common in response to the increasing disparity between processor speeds and disk I/O speeds. Caching disks are disk devices which contain a high-speed memory cache to improve the I/O performance of the disk. To insure that no data is lost, caching disks typically,

[4]Caching disks are sometimes referred to as *intelligent disks*.

but not always, use a non-volatile RAM as the high-speed cache. Due to economic constraints, the caches are usually relatively small (e.g., 32K - 4M bytes [AG91]). In addition to a high-speed cache, the disk control unit contains a microprocessor to manage the cache and control the disk. The microprocessor implements the cache management policy and often uses many of the same caching policies and techniques found in conventional operating systems that perform RAM caching [Gla89]. Despite the intelligent microprocessor in the controller, most caching disks provide a low-level set of disk operations, which may be machine or operating system dependent [Gro85, Pre91, AG91].

Caching disks can be viewed as a special case of the remote memory model. The microprocessor and the cache on the disk provide the hardware functionality required by a memory server, while the bus between the host and the disk serves as the communication channel. However, because each disk physically connects to a single host, caching disks represent a restricted form of the remote memory model. Consequently, only the host directly connected to the disk uses and benefits from the memory resource (cache) on the disk as opposed to the general form of the remote memory model where multiple clients share the large memory resource provided by the memory server. In this restricted form of the model, the disk's cache memory only caches disk I/O data as opposed to providing general purpose data storage. Consequently, the disk memory may be under-utilized and could be better used by the host (as noted in [AG91]). In addition, the system does not support data sharing between hosts as allowed by the remote memory model, nor does it offload backing store activity to allow separate caching optimizations for file system activity and backing store activity.

Because caching disks represent a special case of the remote memory model, many of the memory server techniques could be applied to the cache controller, which acts as a memory server. Using the Logical Memory Server abstraction, caching disks could present a high-level, machine and operating system independent interface to the host machine, thereby allowing use of the disk with a wide variety of operating system and host architectures. The high-level operations of the abstraction, in particular the deactivate operation, would allow the cache controller to quickly reclaim large amounts of cache memory to improve the performance of the cache.

Disk arrays offer another solution to the disk I/O bottleneck [PGK88]. Disk arrays distribute data across multiple disk drives to improve disk I/O throughput. Although disk arrays promise high data transfer rates, the average delay resulting from a seek operation is approximately the same as the average seek time of a conventional disk.

6.6 Summary

This chapter describes the design of a remote memory backing store. We define an abstraction called a Logical Memory Server, which hides the memory server's internal structure from the clients and provides a high-level interface with a well-defined set of operations. The flexibility of the abstraction allows the server to support heterogeneous client machines with a wide variety of operating systems, virtual memory space organizations, page sizes, and byte orders.

To improve client performance, the server uses a hash table and a double hashing algorithm to retrieve data, on average, in constant time regardless of the number of clients, the number of processes, or the amount of data stored on the server. To provide fair memory allocation, the server divides memory into fixed-size data blocks and assigns one or more blocks to each page. Instead of preallocating memory to each client, the server dynamically assigns data blocks to clients based on their needs, thereby sharing the memory resource among all the client machines.

The server uses a transparent, two-level memory scheme to provide an arbitrarily large amount of remote backing store. It uses secondary storage and a memory replacement policy to enlarge the storage capacity of the server.

Reclaiming memory owned by a terminated process or client can consume a substantial portion of the server's processing time and delay processing of new requests. The memory server uses timestamps to process terminate requests in constant time. Each time the server receives a terminate request, it updates the timestamp and immediately sends a terminate reply message. Updating the timestamp invalidates all pages owned by the terminated process. The double hashing algorithm and a garbage collection thread later reclaim the memory occupied by invalid pages. This delayed memory reclamation scheme allows the server to amortize the cost of memory reclamation over time.

7. A PROTOTYPE REMOTE MEMORY MODEL SYSTEM

This chapter describes a prototype system based on the remote memory model. The prototype implementation verifies the design decisions presented in the previous chapters and demonstrates the viability of systems based on the remote memory model. In particular, the prototype provides a basis for investigating the performance of systems based on the model.

Because the previous chapters already described the design, this chapter concentrates on the details of the prototype implementation. The chapter also presents measurements obtained from the prototype.

7.1 System Configuration

We investigated the viability of systems based on the remote memory model by implementing a prototype system based on the model. Figure 7.1 illustrates the prototype VM Xinu system configuration. The prototype system consists of heterogeneous client machines, a memory server, a remote file server, a time server, and a name server. The client machines consist of diskless Sun 3/50, MicroVAX II,[1] VAXstation 3100, and DECstation 3100 workstations. The remote file server executes as a user-level application on a UNIX host and provides the diskless clients with access to the disk storage on the UNIX host. The time server and the name server execute on a UNIX host and provide the standard TCP/IP time service and domain name service. Client machines use the time server and name server to obtain the current time and their domain name at boot time. The memory server executes as a user-level application on a UNIX host. The memory server avoids the use of vendor-specific UNIX

[1]VM Xinu originally ran on a MicroVAX I, the predecessor to the MicroVAX II.

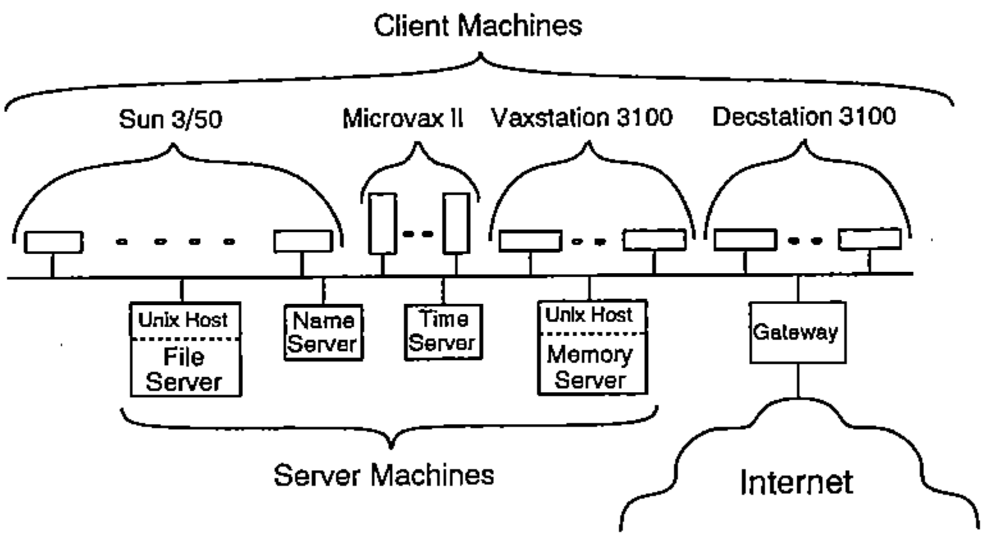

Figure 7.1 The Prototype System Configuration.

system calls and can run on a wide variety of UNIX variants (e.g., SunOS, 4.3BSD, Dynix, System V, or Ultrix). Consequently, we have run the memory server on a Sun 3/50, VAX 11/780, 8 processor Sequent Symmetry, VAXstation 3100, DECstation 3100, and SPARCstation 1+. A 10 Mbps Ethernet connects all the machines and provides access to a gateway that routes packets to the TCP/IP Internet. All the client machines execute the Xinu Virtual Memory Operating System (VM Xinu) and page remotely to the memory server.

Although the clients all execute the VM Xinu operating system, the client architectures differ substantially. The DECstation has a RISC instruction set while the MicroVAX, VAXstation, and Sun have CISC instruction sets. The MicroVAX, VAXstation, and DECstation use reverse byte order (little-endian) while the Sun 3/50 uses forward byte order (big-endian). However, the most notable difference pertains to the virtual memory support provided by the hardware. Each architecture supports a machine-dependent page size, page table entry format, page table organization, and MMU access methods. In particular, the MicroVAX and VAXstation have a 512 byte page size, the DECstation a 4 Kbyte page size, and the Sun 3/50 an 8 Kbyte page size. The clients also differ in their physical memory sizes. The MicroVAX clients used in the prototype system each contain 2MB of physical memory, the Sun 3/50 and VAXstation machines 4MB of memory, and the DECstations 8MB of memory.

7.2 VM Xinu: The Client Operating System

7.2.1 Overview

The VM Xinu operating system [CG91] executes on the diskless client machines and uses remote memory for backing storage. For file storage, the system accesses a remote file server. VM Xinu contains primitives that provide memory management, thread management, thread coordination/synchronization, interprocess communication, real-time clock management, device drivers, and network communication.

VM Xinu supports user-level applications that execute in non-privileged mode and trap to the kernel to invoke system calls. The operating system dynamically loads user-level applications from the file system similar to other operating systems. As we mentioned in chapter 4, the kernel supports multi-threaded user-level applications. All the threads of an application share the text, data, and heap but have separate stacks. In addition, VM Xinu supports multiple threads of control within the kernel. All kernel threads share the kernel text, data, and heap and execute in privileged mode. The lightweight kernel threads carry out various kernel functions in parallel such as page reclamation, network management, timer management, and background paging.

VM Xinu associates a scheduling priority with each thread, not each address space. The system uses a single-level scheduling policy that chooses the next thread to execute from all the threads in the system as opposed to a two-level scheduling policy that schedules address spaces and within an address space schedules threads for execution. The scheduler chooses the highest priority thread from all the threads in the system as the next thread to execute and uses round-robin scheduling for threads with the same priority.

7.2.2 The Virtual Memory System

VM Xinu implements the virtual memory system described in chapter 4. In addition, the virtual memory system defines the mapping from Xinu's virtual memory and process structure to the structured memory space provided by a Logical Memory Server (LMS).

VM Xinu uses the *thread* and *address space* abstractions described in chapter 4 to separate the lifetime of the data from the computation on the data. In VM Xinu the data in an address space may persist after the last thread has completed. To map Xinu's virtual memory structure to the LMS, the virtual memory system separates address space data from thread data by maintaining separate page tables for address spaces and threads. In addition, VM Xinu allocates thread identifiers (TID) and address space identifiers (ASID) from two distinct namespaces. More specifically, thread identifiers range from 0 to $N - 1$ for some fixed value of N, while address space identifiers range from N to $2N - 1$. Consequently, each virtual page belongs to an address space or a thread and is uniquely identified by an (ASID, page number) or (TID, page number) pair depending on the owner. When storing or fetching a page to or from the memory server, the virtual memory system uses the (ASID, page number) or the (TID, page number) as the (VS, page number) on the LMS. The TIDs and ASIDs distinguish thread specific data from shared data and insure that they occupy different VSs on the LMS. Figure 7.2 illustrates the mapping function.

7.2.3 Porting

Recall that VM Xinu contains an architecture interface layer designed to ease porting of the system to new architectures. The architecture interface layer in VM Xinu consists of 26 routines that provide a consistent interface to the underlying virtual memory hardware. To port the virtual memory system from one architecture to the next only involves rewriting the 26 architecture interface routines plus modifying a few system initialization routines. Of the 26 routines, 22 routines are used to *set* or *get* various fields of an abstract page table entry and translate to trivial in-line

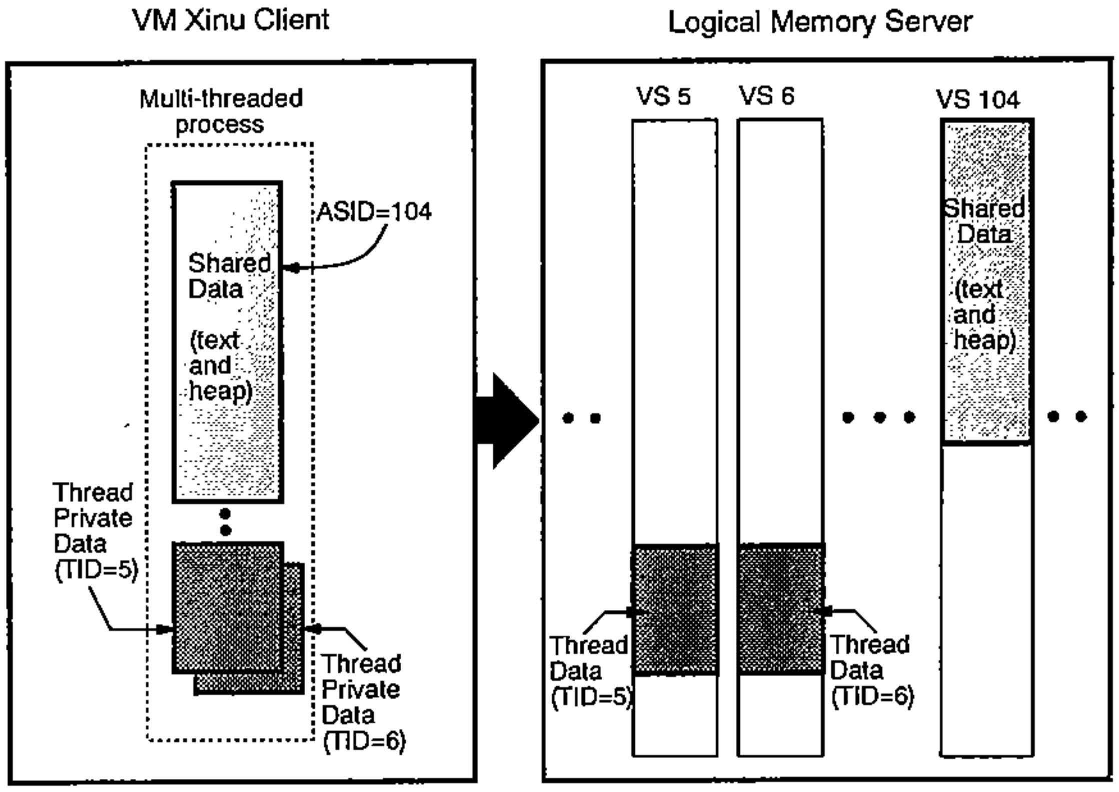

Figure 7.2 The mapping from VM Xinu to an LMS. Xinu's virtual memory system defines the mapping from Xinu's process structure and address space layout to the memory layout of an LMS.

macros on all four client architectures. Only the remaining 4 routines had to be rewritten for each target architecture. Consequently, the architecture interface layer substantially reduced the effort required to port the virtual memory system to the four architectures.

The prototype also demonstrates the ability to support heterogeneous clients. Despite the architectural differences among the client machines, all four client architectures simultaneously access the memory server for backing storage. The memory server presents each client with a Logical Memory Server (LMS) that adapts to the client's architecture. Thus, the memory server supports all three page sizes and byte orders simultaneously.

7.2.4 Unix Emulation

Xinu stands for "Xinu is not UNIX" [Com84]. The differences between Xinu and UNIX prevent Xinu from providing a UNIX-compatible interface. To take advantage of the large amount of Unix software currently available, VM Xinu includes a mechanism that emulates UNIX system calls. Rather than modify the kernel, VM Xinu provides a user-level library that simulates a subset of the UNIX system calls. The library translates UNIX system calls into equivalent VM Xinu calls. The emulation library eases the task of porting existing UNIX software to VM Xinu. To date, the system executes several UNIX applications.[2] More importantly, the emulation library allows us to execute the memory server as a VM Xinu user-level application (see section 7.5.2).

7.3 Implementing The Communication Protocol

To speed the development of a prototype system, VM Xinu uses the User Datagram Protocol (UDP) as the underlying virtual communication channel for the RMCP protocol. Many conventional operating systems provide the support required to access

[2]Current UNIX applications include cat, cp, ls, mv, wc, grep, diff, size, ar, cc, lex, yacc, awk, nroff, mg (an emacs-style editor), the MIT X server [SG86, CCG+91], and various X applications.

UDP from a user-level application. Consequently, using UDP allows us to execute the memory server as a user-level process on a conventional operating system.

Using a virtual network as the underlying communication channel adds a dimension of flexibility not possible with physical network architectures. Physical network architectures only provide communication between hosts directly connected to the network. However, the IP protocol provides communication between all the hosts on a virtual network constructed from multiple physical networks. Because RMCP uses the UDP protocol, clients and servers need not reside on the same physical network. UDP allows clients to page across one or more gateways to memory servers on distant physical networks. In addition, UDP runs on a wide variety of network architectures, allowing us to easily port the system to new network architectures.

Unfortunately, the system pays a price for this added flexibility. Traversing the layers of the TCP/IP protocol suite adds a substantial delay to the time spent transferring a message (see section 7.5.4). In practice, the clients and server often reside on the same physical network and can obtain a significant performance improvement by running RMCP over the raw physical network.

7.4 The Prototype Memory Server

The prototype memory server executes as a user-level application on a UNIX host. The server uses a minimum number of vendor-specific UNIX system calls to simplify the task of porting the server to new UNIX variants executing on new architectures. The ability to execute the server on multiple architectures allows us to measure the effect the memory server's architecture has on performance. In addition, executing the server as a user-level UNIX application allows us to quickly prototype and test new versions of the memory server using the standard UNIX profiling and debugging tools. The server executes in a UNIX environment, as opposed to the native hardware, eliminating the need to implement network device drivers or disk device drivers. The server has a file system available to obtain server configuration parameters at run-time from an initialization file or to log debugging/error messages.

Unfortunately, executing the memory server as a user-level UNIX application has some drawbacks. In particular, the overhead of UNIX impacts the performance of the memory server in several ways. Each system call issued by the server results in a trap to the kernel, which is a relatively expensive operation. Crossing the user/kernel boundary involves changing the address space mappings, switching stacks, and copying the parameters, or data pointed to by the parameters, into the kernel. For example, when sending or receiving UDP messages, the server issues a read or write system call which traps into the kernel and copies the message into a kernel buffer before proceeding. Executing the server as a stand-alone program on the native hardware would avoid such overhead. In addition, the memory server must compete with all other UNIX processes for system resources. Instead of a dedicated machine executing a stand-alone memory server, the server executes concurrently with other UNIX processes and experiences delays due to context switching. Moreover, the memory server may be swapped out like any other UNIX process. Our experience with the prototype shows that clients occasionally experience a long delay while waiting for the operating system to swap in the memory server after a long time of inactivity.

Despite these disadvantages, UNIX provides a nice platform for implementing a prototype server and provides the same functionality as a dedicated machine executing a stand-alone memory server program, albeit slower. Moreover, some of the UNIX overhead can be avoided by carefully designing the tests and experiments used to measure performance.

7.4.1 Internal Data Structures

Each memory server supports multiple Logical Memory Servers (LMS), one per client machine as described in chapter 6. The communication protocol and the memory servers both use a 32-bit LMS ID to identify a particular LMS. Similarly, the protocol and the server both use 32-bit VS IDs and Page Numbers. Consequently, each LMS consists of 2^{32} Virtual Segments each containing 2^{32} pages. That is, the size of the logical storage space presented to each client is $2^{32}*(2^{32}*LMS\_PAGE\_SIZE)$.

To simplify the implementation, the memory server capitalizes on the large virtual address space provided by UNIX. Instead of implementing its own replacement algorithms to move data between memory and secondary storage, the server lets UNIX move data from memory to disk. The server pretends the virtual memory space is really a huge physical memory. The server lets the virtual memory system in UNIX handle all data transfers between physical memory and the swap area.

Allowing UNIX to implement the two-level storage space simplifies the implementation, but places some limitations on the prototype server. In particular, the memory server subjects itself to UNIX's replacement algorithm instead of choosing a replacement policy of its own. Moreover, the server has no means of *locking* pages in memory. Consequently, all the server's data structures, including the hash table, are subject to replacement. Also, because the server does not manage the swap area, it cannot optimize data placement on the disk.

The server divides the memory into fixed-size data blocks and assigns them to clients on demand. Based on our experience with the prototype, we chose to use 1K byte data blocks. Larger block sizes result in a low utilization and considerable internal fragmentation when storing 512 byte MicroVAX pages, while smaller block sizes significantly increase the data block overhead. When the server receives a page_store request for a page larger than 1K bytes, the server allocates multiple physical data blocks to create a logical block large enough to store the page. The internal fragmentation resulting from smaller pages such as the MicroVAX's 512 byte pages can be reduced by storing two consecutive pages of a VS per data block (i.e., consolidate two consecutive VS pages into a logical page). As a result of the locality of reference principle, client processes tend to access consecutive pages of their virtual address spaces, which the virtual memory system eventually stores in consecutive pages of a VS. Consequently, consolidating two consecutive VS pages into a logical page often results in efficient use of data blocks for page sizes less than the data block size.

The memory server described in chapter 6 uses a garbage collection thread within the server to reclaim invalid pages. Unfortunately, standard UNIX does not support multi-threaded processes. Consequently the memory server simulates a background garbage collection thread using signals and alarms, and checks for new requests using the select() call. If there are no requests when the alarm expires, the process executes the garbage collection routine for a short period before checking for paging requests again.

Although the remote memory model provides the opportunity for clients to share data, the current prototype does not support any data sharing between client machines.

7.4.2 Double Hashing

The memory server uniquely identifies each page using a (LMS ID, VS ID, page number) triple. When the server receives a page store or page fetch request, it applies a double hashing algorithm to the triple to locate the hash table entry containing the page.

Double hashing takes a *key* and produces an index into the hash table. Consequently, the server must first reduce the triple to a single number that can be used as the key. To obtain a single number, the server applies the *folding* function [HS82] shown in Figure 7.3.

```
Fold(lms, vs, pn)
    Bit32 lms, vs, pn;
{
    Bit32 t1, t2, t3, value;           /* 32 bit integers          */

    t1 = lms << 26;                    /* LMS ID upper 6 bits      */
    t2 = vs << 16;                     /* VS ID middle 10 bits     */
    t3 = pn;                           /* Page Number low 16 bits */
    value = ((t1 ^ t2) ^ t3);          /* XOR all three            */
    return(value);
}
```

Figure 7.3 The memory server folding function.

Because the folding function compresses 96 bits of information into a 32 bit key, some information will be lost. Consequently the folding function should uniformly distribute the triples across all possible keys to reduce the number of collisions. The folding algorithm above assumes that LMS IDs and VS IDs are relatively small numbers and therefore only uses the low order bits of the LMS ID and the VS ID. Because the server allocates small LMS IDs first, it is reasonable to assume small LMS IDs. Because VM Xinu uses the thread ID or address space ID as the VS ID, the resulting VS IDs range from 0 to $2N$, where N is the maximum number of address spaces allowed on the client. Consequently, is it reasonable to assume small VS IDs. Clearly, the folding function optimizes for the VM Xinu system but should be modified if used with other systems.

Once the server has compressed the triple into a single key, the server applies a double hashing algorithm to the key. Figure 7.4 illustrates the double hashing algorithm used to locate the desired hash table entry in response to a fetch request. To reduce the average number of probes required to locate an entry, the server uses a hash table size of N, where N and $N-2$ are twin primes [Knu73]. During each probe to the hashing table, the algorithm checks the timestamp of the entry and reclaims the entry if the timestamp has expired.

The server also uses a double hashing algorithm to add entries to the LMS and VS hash tables. However, the VS hash table only needs to fold the LMS ID and the VS ID into a single key. The LMS hash table does not require any folding.

7.5 Experimental Results

The first and most basic test of the implementation involved setting up the system pictured in Figure 7.1 and allowing the heterogeneous clients to page to a single memory server. We executed the VM Xinu operating system on all four architectures simultaneously to demonstrate memory server support for heterogeneous client machines.

```
/* Double Hashing Algorithm                               */
/* Chose N such that N and N-2 are twin primes            */


define hash1(key)        (key mod N)
define hash2(key)        ((key mod (N-2)) + 1)

Double_Hash(key, triple)
{
    current_entry = hash1(key)
    index = hash2(key)
    entry_not_found = TRUE
    unchecked_entries_left = N
    while (entry_not_found and unchecked_entries_left) {
        if (current_entry is occupied) {
            check the entry's timestamp against
                the LMS and VS timestamps
            if (timestamps differ) {
                reclaim invalid entry
            }
            else if (current_entry matches triple) {
                entry_not_found = FALSE
                location = current_entry
            }
        }
        current_entry = (current_entry + index) mod N
        unchecked_entries_left = unchecked_entries_left - 1
    }
    if (entry_not_found)
        return error
    else
        return location
}
```

Figure 7.4 The double hashing algorithm for a fetch_request. The algorithm assumes a hash table of size N.

The remainder of this section describes experiments designed to measure the performance of the prototype. This section also describes the measured performance of an existing distributed system for the purpose of comparison.

7.5.1 Conventional Backing Store Performance

To establish a point of reference, we measured the performance of a conventional virtual memory system used in a production environment on a daily basis. In particular, we measured the performance of a diskless Sun 3/50 machine executing SunOS 4.0 and paging across a 10 Mbps Ethernet to a Sun 3/50 NFS file server. The SunOS/NFS model is similar to the remote memory model in that they both support diskless nodes paging to a remote backing store. To reduce the influence of external factors, we ran the tests in the evening while the file server and the network were both lightly loaded.

Several studies have reported on the performance of Sun's NFS distributed file system [LS89, SGK+85, Kei90]. However, our interest focused on the performance of NFS as a backing store for diskless nodes. The majority of NFS requests consist of directory lookup operations and get/set attribute operations [LS89]. Our measurements factor out all but the virtual memory system paging activity. In particular, we measured the average round trip delay to store or fetch an 8K byte page to or from an NFS server. The measurements were obtained by modifying the SunOS kernel to record the round trip delay for each paging request.

Table 7.1 Average round trip delay to store or fetch an 8K byte page to or from a Sun 3/50 NFS file server from a diskless Sun 3/50 client running SunOS.

Operation	Time
Fetch	84 ms
Store	176 ms

Table 7.1 shows the average round trip delay experienced by SunOS when transferring data to or from the file server. The table shows that NFS adds a substantial delay to ensure that each store operation commits the data to disk before acknowledging the operation.

To compare against systems with a local disk, we measured the performance of a Sun 3/50 executing SunOS 4.0 paging to a swap partition on a standard Sun 3/50 disk drive.[3] Again we ran the tests on a lightly loaded system. Unlike the diskless client paging to NFS, the average time to store data on the disk was approximately equal to the time required to fetch data from the disk.

We executed two tests. Each test measured the average disk transfer rate of the virtual memory system during the execution of the test program. Both test programs allocated a heap region much larger than the size of the physical memory. The programs then proceeded to touch all the pages in the newly allocated space, thereby causing the virtual memory system to page to the local disk. The first program sequentially traversed the space, touching one byte of each 8K byte page. The second program randomly touched one byte of each 8K byte page. The first test represents an extreme case of the *locality of reference* principle, referencing all the pages in order. The random test illustrates the other extreme. A typical system's behavior lies somewhere in between. Table 7.2 shows the average 8K byte disk transfer times

Table 7.2 Average delay that results from storing or fetching an 8K byte page to or from a local disk drive on a Sun 3/50 executing SunOS.

Test Program	Time
Sequential VM access	22 ms
Random VM access	30 ms

[3]A Micropolis 1325 disk drive.

for the two programs. The measured performance shown in Table 7.2 agrees with the local disk performance reported in the literature [Hab89, Wil89].

7.5.2 Remote Memory Backing Store Performance

We performed several tests using the prototype system that measured various aspects of the system's performance. In order to compare the performance of the prototype system against performance of the previous systems, we used a configuration similar to the diskless SunOS/NFS system for which the performance measurements were reported in section 7.5.1. We measured the performance of a diskless Sun 3/50 executing the VM Xinu operating system and paging across a 10 Mbps Ethernet to a Sun 3/50 memory server. Again we ran the tests during the evening when the server and the network were lightly loaded. Furthermore, to keep the results as comparable as possible, we executed the memory server as a user-level process on the same file server machine used in the SunOS test. To insure the entire memory server remained resident in memory, we artificially limited the amount of data clients stored on the server to the memory server's resident set size.

Table 7.3 compares the performance of a diskless VM Xinu client to the performance of the diskless SunOS system. Although the memory server executes as

Table 7.3 Average round trip delay for VM Xinu paging to a memory server vs. SunOS paging to NFS. Both systems used a Sun 3/50 client and a Sun 3/50 server.

Operation	VM Xinu	SunOS/NFS
Fetch	31 ms	84 ms
Store	31 ms	176 ms

a user-level process, the VM Xinu results show a significant improvement over the SunOS system paging to NFS. In particular, the results show that the time to service a page fault in SunOS takes almost three times as long as the time to service a page fault in VM Xinu. NFS commits each store operation to disk before acknowledging the operation. Consequently, VM Xinu store times are 5 times faster than SunOS store times. Unlike SunOS, the store and fetch times in VM Xinu are symmetric.

Using the UNIX emulation library in VM Xinu, we executed the sequential and random access tests described in the previous section. Table 7.4 compares the performance of VM Xinu paging to a memory server to the performance of SunOS paging to a local disk. In the case of the random test, VM Xinu demonstrates performance

Table 7.4 Average round trip delay for VM Xinu paging to a memory server vs. SunOS paging to a local disk.

Test Program	VM Xinu	SunOS/Disk
Sequential VM access	31 ms	22 ms
Random VM access	31 ms	30 ms

competitive with SunOS paging to a local disk. The sequential test represents the other extreme and shows that remote memory performance is not more than 1.5 times slower than a local disk.

7.5.2.1 Remote Memory Access Times on Various Hardware Configurations

Because the memory server executes as a user-level UNIX process, we were able to execute the memory server on a wide variety of architectures and UNIX variants. This flexibility allowed us to measure the effect the architecture and operating system have on performance. Table 7.5 shows the average round trip delay for paging requests when the memory server executes on various operating systems and hardware architectures.

Table 7.5 The time to store/fetch an 8K byte page when the memory server executes on various UNIX systems. In each case, the clients consist of Sun 3/50 machines executing VM Xinu.

	Sun 3/50 (SunOS)	Sun 3/50 (VM Xinu)	Sparc 1+ (SunOS)	Vaxstation 3100 (Ultrix)	Decstation 3100 (Ultrix)
Store	31 ms	31 ms	20 ms	29 ms	21 ms
Fetch	31 ms	31 ms	23 ms	33 ms	28 ms

The table shows the results of five independent tests. All five tests measured the performance of a single Sun 3/50 VM Xinu client paging across a 10 Mbps Ethernet to a memory server executing on a UNIX host.[4] Each test used a new architecture and operating system as the UNIX host. The top row of the table indicates the architecture and operating system used as the memory server. Each column of the table shows the time required to store or fetch an 8K byte page to or from the memory server. The results show that the architecture and operating system significantly affect the performance of the memory server. Clearly, a memory server running on one of the faster machines (a DECstation 3100 or Sparcstation 1+) outperforms a memory server running on one of the slower architectures (a Sun 3/50 or VAXstation 3100). Notice that the RISC machines (the DECstation 3100 and the SPARCstation 1+) provide backing storage at speeds competitive with the local disk speeds reported in Table 7.2.

Note that the time required to store a page differs from the time required to fetch a page when paging to a Sparcstation 1+. However, when paging to a Sun 3/50, the times are symmetric. This difference results from the design of the network layer in VM Xinu which uses a kernel thread to demultiplex incoming network packets. The kernel thread waits for an incoming packet, demultiplexes the packet based on its packet type, awakens any processes that may be waiting for the packet, and then

[4]Except for the test which ran the server on a VM Xinu host using the UNIX emulation library.

waits for the next network packet to arrive. Using a kernel thread to process incoming packets simplifies the design; however, the additional context switching that results from using a kernel thread increases the time required to read a multi-packet message. Measurements indicate that the additional context switching, combined with a non-optimized Ethernet interrupt handler, adds approximately .45 ms to the processing time for each incoming packet on a Sun 3/50. Consequently, VM Xinu processes an incoming 8K byte message approximately 2.7 ms slower than an 8K byte outgoing message. However, this difference does not appear when paging to SunOS on a Sun 3/50 because VM Xinu still reads packets faster than SunOS on a Sun 3/50 can send packets.

In the case of the two Ultrix systems, the networking delay introduced by VM Xinu does not account for the entire difference between the store times and the fetch times. The high fetch times shown for the two Ultrix systems result from Ultrix's implementation of the UDP/IP protocols. Ultrix sends packets substantially slower than it receives packets. Consequently, the store times are much lower than the fetch times. SunOS, on the other hand, sends and receives packets at roughly the same speed. Clearly, the operating system's implementation of the UDP/IP protocols has a significant effect on paging times.

The second column of the table shows the average round trip times for a memory server executing as a VM Xinu user-level process. Since Xinu is not UNIX, the server spends a significant amount of time in the UNIX emulation library. In addition, VM Xinu has not had the man-years of fine tuning and optimizations that SunOS has had. Consequently one might expect SunOS to outperform VM Xinu. However, the table shows that performance is the same for both systems. The additional time spent emulating UNIX is offset by the fact that VM Xinu sends and receives network packets faster than SunOS.

Having evaluated the effect various server architectures have on paging times, we measured the performance of VM Xinu executing on a DECstation 3100 to determine the impact a different client architecture has on paging times. Table 7.6 shows the paging times for a DECstation 3100 paging across a 10 Mbps Ethernet to two memory server architectures. The DECstation 3100 has a 4K byte page size. Consequently, the

Table 7.6 Paging times from a DECstation 3100 executing VM Xinu. The time shown is the round trip delay to store/fetch a 4K byte page to or from a memory server executing on the specified architecture.

	Sparcstation 1+ (SunOS)	Decstation 3100 (Ultrix)
Store	8 ms	8 ms
Fetch	10 ms	12 ms

times in Table 7.6 reflect the round trip delay to fetch or store a 4K byte page. Again, the difference between the store and fetch times reflect the VM Xinu context switching overhead and the Ultrix overhead. Clearly, the faster processor on the DECstation 3100 and the smaller page size result in average access times substantially better than the slower Sun 3/50 client architecture and significantly better than most disk drives. In the remote memory model a smaller page size significantly reduces the average access time. However, the average seek time on a local disk does not depend on the size of the data transferred. Consequently, a smaller page size does not significantly reduce the average access time to a local disk. Both Table 7.5 and Table 7.6 indicate that the remote memory model capitalizes on current trends in computer technology. As network bandwidth, CPU speeds, and memory sizes increase, the performance of remote memory backing storage will continue to improve.

7.5.3 Memory Server Performance

7.5.3.1 Client Load

To determine the performance of the memory server under various loads, we measured the average response time of a Sparcstation 1+ memory server supporting multiple Sun 3/50 VM Xinu client machines. We generated the various server loads by varying the number of clients and the number of requests per second issued by each client. Figure 7.5 illustrates the memory server response time for various server loads. The figure plots the average response time when executing one, three, and six clients. At 100 requests per second the prototype server becomes saturated. Any additional load on the server results in queueing delays that significantly increase the round trip time. However, for three or less clients, the figure shows that the average round trip delay remains relatively constant, regardless of the load on the server. For six clients, the average round trip delay remains constant for low loads, and then begins to rise as the load increases. However, the increased load on the server does not account for the increase in the average round trip delay (as evidenced by the three client line). Instead, the increase results from contention for the network and the queueing delays experienced by requests that arrive simultaneously from multiple machines. Note that the collective throughput remains the same despite the higher round trip delays. That is, each client still receives $\frac{1}{n}th$ of the total throughput, where n is the number of clients. In the case of six clients, each client obtains a throughput of approximately 1 Mbps. Consequently, as long as the combined request rate of the client machines remains less than the saturation rate, new clients can be added without degrading the server's performance (i.e., the server will continue to perform at its maximum processing rate [e.g., 100 requests per second], regardless of the number of clients using the server).

Note that at 100 requests per second the memory server processes 800K bytes of data per second and consumes more than 6.25 Mbps of the Ethernet bandwidth. Thus, neither the network bandwidth nor the server's processing rate present a major

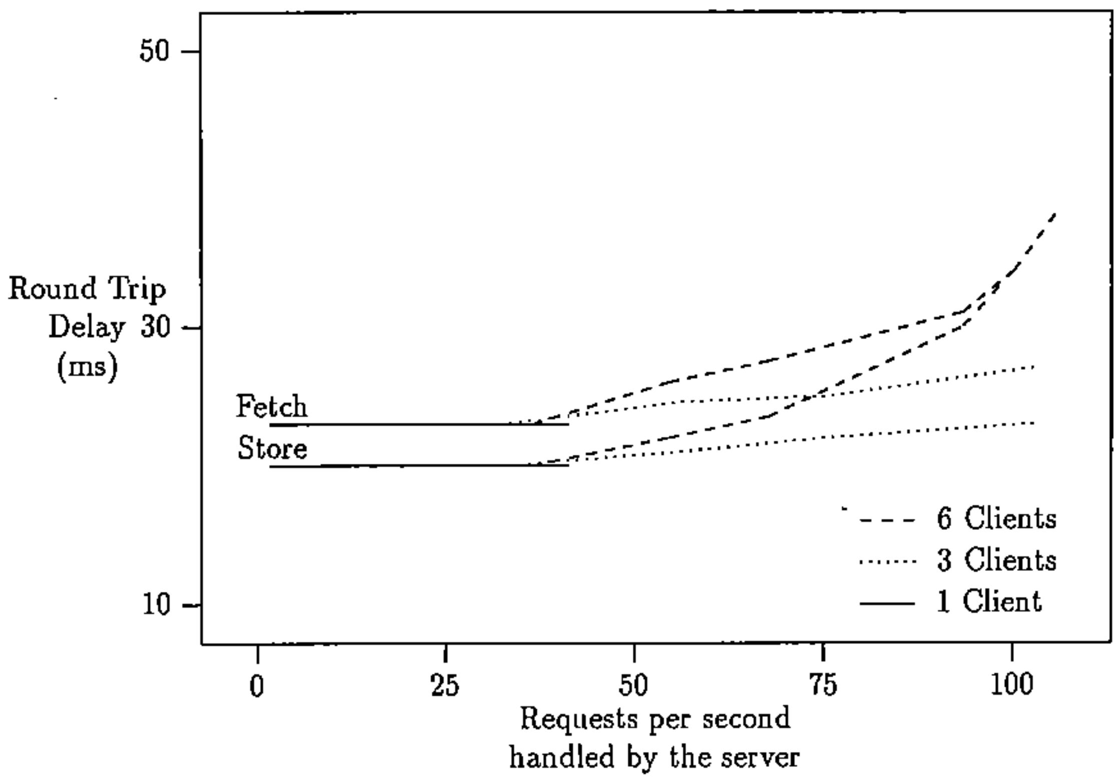

Figure 7.5 Average round trip delay to store or fetch an 8K byte page as a function of the server load. The times indicate the average round trip delay experienced by Sun 3/50 client machines paging to a memory server executing on a SPARCstation 1+. Note that at 100 requests per second the server handles 800K bytes of data per second and consumes 6.25 Mbps of the Ethernet bandwidth.

bottleneck. At peak performance, each executes near, or at, its saturation point. Assuming that network speeds, CPU speeds, and network I/O devices increase in performance at the same rate, the results obtained from current technology indicate that the system will perform well in the future.

In a conventional virtual memory system (i.e., one which uses magnetic disks for backing storage), the average seek time of the disk is the bottleneck. However, in the remote memory model, contention for the network and contention for the server present the major bottlenecks. Contention refers to the number of clients that simultaneously attempt to access the server. Contention reduces the throughput attainable by each client and may increase the average round trip delay experienced by a client. Each remote memory model system can only tolerate a certain amount of contention before the performance exceeds some minimum performance threshold. Although the maximum allowable contention is dependent on the requirements of the system, the results shown in Figure 7.5 indicate that systems in which there are at most six machines contending for the server at any given time will exhibit performance competitive with systems that page to a local disk. In addition, our experience indicates that virtual memory systems often transfer data in bursts (e.g., swapping out an idle process) followed by long periods of inactivity. The bursty behavior helps reduce contention by reducing the number of clients simultaneously accessing the memory server. Consequently, the number of clients using a memory server may be several orders of magnitude larger than the contention level for the server.

7.5.3.2 Data Retrieval

To measure how well the server scales as the amount of data stored on the server increases, we measured the performance of the server's lookup algorithm as a function of the amount of data stored on the server. Theoretically, the double hashing algorithm provides constant time access to the data as long as the hash table remains less than 95% full. However, in practice, the implementation uses a folding function in addition to a double hashing algorithm. The server folds the triple identifying a page into a single key before applying the double hashing algorithm.

Figure 7.6 shows the average number of probes required to locate a page in the server's hash table as a function of the hash table utilization. We modified the memory

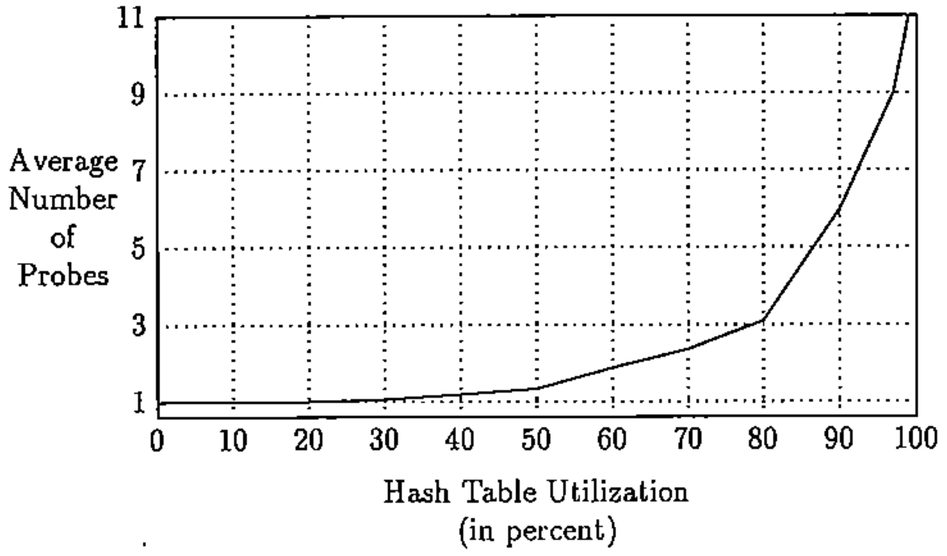

Figure 7.6 Measured performance of the double hashing algorithm as a function of the hash table utilization.

server to record the average number of probes per request as a function of the number of hash table entries in use and then executed several client machines all paging to the memory server. Because UNIX only allows the memory server to obtain a fixed amount of heap space for data storage, we artificially limited the size of the hash table to 1021 entries to allow 100% utilization of the hash table[5] (i.e., we did not want the hash table to map more storage space than the server could obtain). The figure shows the combined performance of the folding function and the double hashing algorithm. Even when the hash table is 90% full, the server only requires an average of 6 probes to locate the desired data. As long as the server limits table utilization

[5]1021 is a twin prime and corresponds to approximately 8M bytes of storage space for Sun 3/50 clients.

to 90%, the average access time remains constant (i.e., $O(6)$). However, at 80%, the server requires only 3 probes to locate the desired entry. Thus, a small increase in the amount of memory consumed by the hash table (i.e., enlarging the hash table to lower hash table utilization) results in a substantial increase in performance (i.e., a factor of 2 improvement in performance).

7.5.4 Protocol Performance

The paging times reported in Table 7.5 measure the combined performance of the remote memory server and the communication protocol. Our experience with the system indicated that the communication protocol, rather than the memory server, constituted the majority of the delay associated with accessing data on the server.

Because NAFP hides the underlying communication channel from the XPP protocol, the communication protocol runs over most network architectures. In our prototype implementation, we built the communication protocol on top of the UDP protocol, using the virtual network provided by the IP protocol to allow access to memory servers on remote networks. Because our design does not bind the communication protocol to any particular communication channel, we wanted to measure the additional overhead incurred as a result of our implementation decision to use UDP as the communication channel rather than a link-level communication channel (e.g., an Ethernet).

Table 7.7 shows the breakdown of an 8K byte paging request from a Sun 3/50 client paging to a Sun 3/50 memory server in terms of the time spent processing each stage of the request. The UDP/IP component includes the time spent transmitting the data across the wire. However, because the client's CPU and transceiver execute concurrently, the transmission time for fragment i overlaps the XPP/NAFP and UDP/IP processing of fragment $i + 1$. Consequently, the table only counts the overlap processing once. The table shows that the majority of the paging time can be attributed to the UDP/IP protocol. Consequently, building the communication

Table 7.7 A breakdown of the time required to process an 8K byte request from a Sun 3/50 client paging to a Sun 3/50 memory server. The percentage is calculated from a total paging time of 31 ms.

Component	Time	Percentage
XPP/NAFP	8.6 ms	28%
UDP/IP	15.6 ms	50%
Memory Server	6.8 ms	22%

protocol directly on the underlying physical network (in this case the Ethernet) would substantially improve paging performance.

7.5.4.1 Data Streaming

Our experience indicates that most virtual memory systems exhibit bursty backing store I/O behavior. Under normal processing demands the virtual memory system rarely pages to the backing store. However, when a user executes a program requiring a large amount of memory the virtual memory system suddenly transfers large amounts of data to the backing store. Also, whenever the user resumes a process that has been swapped out, the virtual memory system suddenly transfers a large amount of data between memory and the backing store. The long periods of infrequent paging interspersed with sudden flurries of paging activity emphasize the need for data streaming.

Although data streaming allows the virtual memory system to use more of the network bandwidth, choosing an optimal pending list length is difficult and depends on several factors.[6] One might think that increasing the pending list length would always improve the client's performance because the virtual memory system can use more of the network bandwidth. However, Figure 7.7 indicates that our intuition

[6]Chapter 4 and chapter 5 describe the role of the pending list. Briefly stated, the pending list length specifies the maximum number of requests a client may issue before receiving a reply.

is not necessarily correct. Figure 7.7 shows the total runtime (in seconds) of the

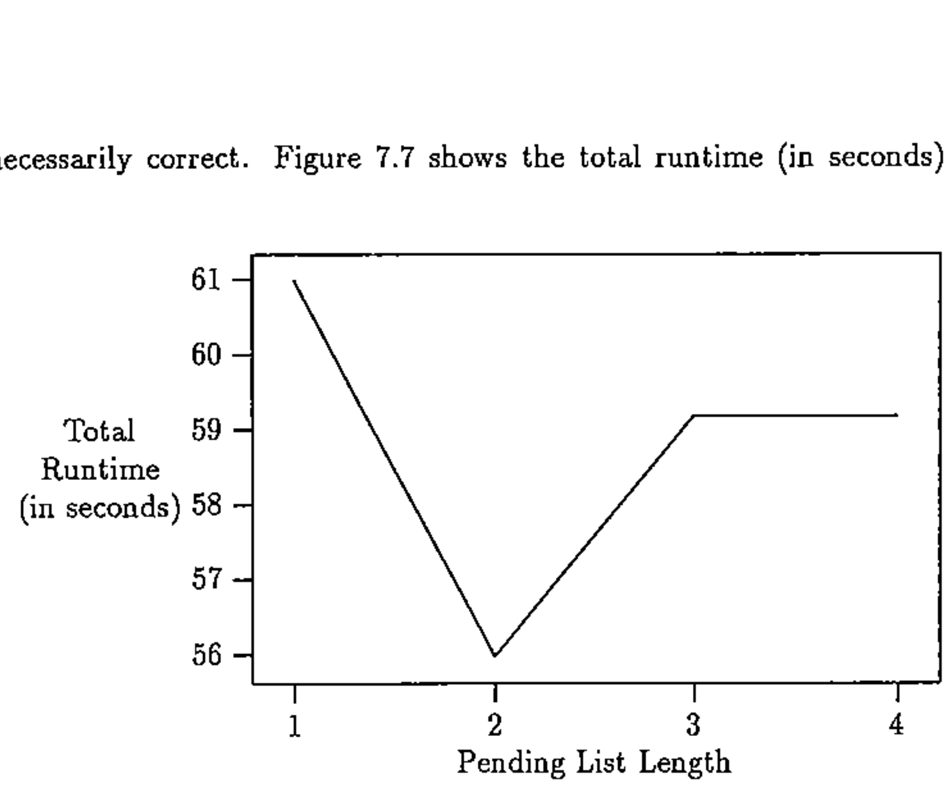

Figure 7.7 The total run time of the sequential access test program as a function of the pending list length.

sequential access test described in section 7.5.1 for various pending list lengths. The total runtime decreases when we change the pending list length from 1 to 2; however, the total runtime increases when we go to a pending list length of 3. To understand the anomaly in the graph, one must examine the application (in this case the sequential access test) and the behavior of the page replacement algorithm.

The sequential test program incurs page faults at a steady and rapid rate, performing almost no computation between faults. Consequently, the number of free pages drops to the low water mark and the replacement algorithm executes continuously, trying to reclaim memory. When the pending list length is 1, the virtual memory system synchronously issues a fetch request in response to a page fault. As soon as the fetch request completes, the virtual memory system issues a store request to replenish the free list. While waiting for the store request to complete, the test program faults again, and the cycle continues (see Table 7.8). With a pending list of length 2, the virtual memory system issues a fetch request and a store request back-to-back.

Table 7.8 Sequence of requests observed at the memory server for various pending list lengths. The letter f denotes a fetch request and the letter s denotes a store request.

Pending List Length	Request Pattern
1	fsfsfsfsfsfs
2	fssfssfssfss
3	fsssfsssfsss
4	fsssfsssfsss

When the client receives the fetch reply it immediately sends a second store request. Because the virtual memory system issues the first store request before receiving the reply to the fetch request, the total runtime decreases. When the pending list length is 3, the virtual memory system asynchronously issues a fetch request followed by two store requests. While processing the second store request, the client receives the fetch reply and issues a third store request. Because the page replacement policy aggressively frees pages, the virtual memory system issues 3 store requests between each fetch request as opposed to the 2 store requests when the pending list length is 2. As a result, the virtual memory system issues many more store requests for a pending list length of 3 than for a pending list length of 2. The type of application, the page replacement policy, the scheduling policy, and the server response time all affect the choice of an optimal pending list length. However, the figure clearly shows that even a small pending list of length 2 substantially improves the client's performance. In this case, data streaming results in an 8.3% improvement in total execution time over synchronous delivery.

7.6 Summary

In this chapter we describe a prototype implementation based on the remote memory model and present experimental results obtained from the prototype.

We implemented and ported the VM Xinu operating system to the Sun 3/50, MicroVAX II, VAXstation 3100, and the DECstation 3100. We implemented the memory server as a user-level application executing on a UNIX host and used a 10 Mbps Ethernet to connect all the machines. Executing the memory server as a UNIX application allowed us to quickly prototype and test the memory server using standard UNIX profiling and debugging tools. We demonstrated heterogeneous client support by executing VM Xinu on all four architectures. Although each architecture supports a machine-dependent page size, all four architectures paged to a single memory server simultaneously.

Experimental results obtained from the prototype indicate that the remote memory model offers performance competitive with conventional diskless systems and systems with a local disk. Our results show that the average time required to store or fetch an 8K byte page to or from a local disk ranges between 22 ms and 30 ms. In comparison, our prototype system averages 20 ms to 31 ms to store (fetch) an 8K byte page to (from) the memory server. In addition, the results indicate that the server scales well as the number of requests per second processed by the server increases. Our implementation decision to use UDP as the underlying communication channel significantly impacts the system's performance. However, even with the extra overhead resulting from UDP/IP, the system performs at speeds competitive with a local disk. In addition, UDP allows clients to page across one or more gateways to memory servers on distant physical networks. In short, the prototype clearly demonstrates the viability of designing systems based on the remote memory model and indicates that the model will perform well in the future as CPU speeds, network bandwidth, and memory sizes increase.

8. CONCLUSIONS

This thesis investigated a new model for designing distributed systems based on remote memory backing storage. We described a remote memory model system and a prototype implementation built to demonstrate the viability of the model.

The remote memory model uses dedicated, large memory machines to provide a shared remote memory backing storage resource to a set of heterogeneous client machines. The system achieves high performance through the use of a special purpose communication protocol and a memory server with highly-efficient data structures and algorithms. In addition to providing a highly-efficient backing store, the model allows data sharing among clients, improves file system performance by offloading the file server, and capitalizes on hardware advances that improve network bandwidth, increase CPU speeds, and expand memory sizes.

The remainder of the chapter briefly highlights some of the contributions of the research and suggests directions for future research.

8.1 Hierarchical Design

We presented a virtual memory operating system that incorporates virtual memory and support for remote memory backing storage into a hierarchical operating system design. The hierarchical design partitions operating system functions into distinct components and arranges the components into a layered hierarchy. Not only does a hierarchical design combine closely related operations into layers with well-defined semantics and interfaces, but it also specifies the dependencies between layers. The hierarchical design clearly defines the interaction between the various components, making the system easier to understand and modify. In particular, the layering allows one to easily modify the virtual memory system to support several types of backing storage without affecting other components of the system. VM Xinu supports remote memory backing storage using lightweight kernel threads, shared memory, and efficient interprocess communication. The VM Xinu system demonstrates that remote memory backing storage can be incorporated into a hierarchical design without sacrificing efficiency (i.e., the system exhibits performance competitive with existing systems).

8.2 An Efficient Paging Protocol

To minimize the delay incurred when accessing remote memory backing storage, we designed a special purpose communication protocol with low overhead and low delay.

The communication protocol supports data streaming. Data streaming allows the virtual memory system on the client to use more of the network bandwidth by transmitting new requests before receiving acknowledgements for previous requests. The protocol includes a *preceding message number* to define a partial ordering on the list of messages. The preceding message number allows the memory server to process paging requests out of order and achieve the same results that in-order processing would have produced.

The low-level negative acknowledgement fragmentation protocol (NAFP) allows execution of the communication protocol over any communication channel that provides unreliable delivery. Negative acknowledgements improve efficiency by detecting fragmentation errors as soon as they occur, but add no computational overhead in the absence of errors. In addition, negative acknowledgements can be used as a general purpose technique to improve the performance of almost any high-level protocol, or combination of protocols, that provide fragmentation and reliable delivery.

8.3 Remote Memory Service for Heterogeneous Machines

The memory server provides remote memory backing storage to multiple client machines simultaneously. It uses the Logical Memory Server (LMS) abstraction to hide the internal organization and data storage and support heterogeneous clients.

The memory server uses a single hash table to store the data from all the clients. The hash table acts like a page table and allows the server to present clients with a single-level storage space (i.e., an LMS) but can use a two-level storage space consisting of primary memory and secondary storage to store the data. The single hash table, together with the double hashing algorithm, allow the server to process fetch and store requests, on average, in constant time, regardless of the number of clients using the server. Timestamps allow the server to process terminate requests in constant time and amortize the cost of memory reclamation over time.

8.4 Competitive Performance

We designed and implemented a prototype system based on the model and measured the system's performance. The prototype system demonstrates the viability of systems based on the remote memory model and exhibits performance competitive with existing distributed and stand-alone systems. In particular, the results show that a memory server executing on a conventional RISC architecture such as a DECstation 3100 or a SPARCstation 1+ exhibits performance competitive with, and in some cases better than, a local disk drive. The results also show that the memory server scales well as the number of clients increases. Furthermore, the prototype indicates that the remote memory model will perform well in the future as network bandwidth, CPU speeds, and memory sizes continue to increase.

8.5 Future Work

Although the prototype implementation demonstrates the viability of distributed systems based on the remote memory model, several interesting extensions exist and warrant further investigation.

8.5.1 Data Sharing

The current prototype does not allow data sharing among client machines. To support data sharing among machines, we must develop memory server primitives that allow clients to create and obtain shared data regions on the server.

Several data sharing possibilities exist. Read-only data sharing is simple to implement and avoids coherency problems by only allowing clients to share immutable data. Despite its simplicity, read-only data sharing allows clients to share text images, libraries, and static databases, thereby reducing the amount of memory consumed on the memory server. Another alternative is read-write data sharing. Several forms of read-write data sharing exist, each having slightly different semantics and coherency guarantees[Li86, LH89]. Unlike read-only sharing, read-write sharing may require *callbacks* to ensure coherency[FP89, RK88]. Data sharing mechanisms also specify the granularity of sharable data. For example, the memory server may define the smallest unit of sharable data to be the size of an LMS page, the size of a data block,[1] or the size of an entire Virtual Segment (VS). Typically there is a tradeoff between the efficiency and the functionality supported by the data sharing model. We need to choose a data sharing model that results in the desired combination of efficiency and functionality and then extend the primitives provided by the memory server to support the chosen data sharing model.

[1]The fixed size blocks of physical memory used by the memory server to back the pages of a VS.

8.5.2 Memory Server/File Server Interaction

The current definition of the remote memory model includes a memory server machine and a file server machine. In the future, we plan to investigate the relationship between the memory server and the file server and explore new methods for the memory server and the file server to communicate and interact.

Again, several options exist. For example, the memory server could access the file server directly. The memory server would then allow clients to request that a specific file on the file server be used as the initial contents of a newly created VS. Alternatively, the file server could understand the operations supported by a memory server and allow clients to request that a certain file be installed on the memory server.

Another possibility is to merge the file server with the memory server. Using a memory server as the basis for a remote file system produces several intriguing possibilities. A memory-server-based file system would allow us to separate naming and protection from data storage. In this approach, a *directory server* would provide the naming and protection aspect of the file system and use the memory server for data storage. Adding a *store-and-save* operation to the memory server would allow the file server to use either volatile or non-volatile storage for file system data and provide conventional stable storage or high-speed volatile storage (e.g., temporary files).

Another alternative would be to design an object-oriented file system. Each file would be an object consisting of a virtual address space and a set of operations (e.g., open, close, read, and write). Because each file object is really just an address space and thread(s) of control, the resulting system has a single data storage paradigm: virtual memory backing storage.

We also plan to remove the restriction that pairs each client with exactly one memory server. We want to allow clients to spread data across multiple servers to improve reliability. Clients could also partition the data and then store it on multiple servers to improve performance.

We would also like to investigate new memory server configurations. If a memory server provides better performance than a local disk, then it becomes advantageous to use memory servers to back other memory servers. Chaining memory servers together or arranging them into a hierarchy allows the system to provide an arbitrarily large amount of remote memory storage space.

8.5.3 Communication Protocol

Several aspects of the communication protocol deserve further attention. Section 7.5.4 showed that the application, the page replacement algorithm, the scheduling policy, and the server response time all affect the choice of an optimal pending list length. More investigation is needed to determine the optimal pending list length. Another alternative would be to use a dynamic pending list length similar to the window size in TCP[Com88] but based on the average amount of time a fetch request spends waiting for store requests to complete.

Future communication networks will provide data transfer rates several orders of magnitude faster than conventional networks. Not only will the higher bandwidth affect the performance of the communication protocol, but large bandwidths may mandate modifications to the page replacement algorithm. We would like to investigate the relationship between the communication protocol and the replacement algorithm in light of higher-speed networks.

8.6 Summary

This thesis explored a new model for designing distributed systems called the remote memory model. The remote memory model provides an attractive alternative to conventional memory models. In the remote memory model, memory servers provide high-speed access to a large, shared memory resource that provides backing storage to virtual memory systems executing on a set of heterogeneous client machines. The system uses optimized data structures, algorithms, and a connectionless, data streaming protocol to achieve high performance.

We designed and implemented a prototype system to demonstrate the viability of systems that use remote memory backing storage. The prototype shows that remote memory systems offer performance competitive with, and in some cases better than, conventional virtual memory systems.

Finally, this thesis has advanced our understanding of virtual memory in a distributed system.

BIBLIOGRAPHY

BIBLIOGRAPHY

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazawska, and Jerre D. Noe. The Eden System: A Techincal Review. *IEEE Transactions on Software Engineering*, SE-11:43–58, January 1985.

[ADU71] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of Optimal Page Replacement. *Journal of the ACM*, 18(1):80–93, January 1971.

[AF88] David P. Anderson and Domenico Ferrari. The DASH Project: An Overview. Technical Report UCB/CSD 88/405, University of California, Berkeley, February 1988.

[AG91] Steve Apiki and Rick Grehan. Caching Cards Speed Data Access. *Byte*, 16(1):168–185, January 1991.

[AHJ90] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. Implementing and Programming Causal Distributed Shared Memory. Technical Report GIT-CC-90-49, College of Computing, Georgia Institute of Technology, 1990.

[Ame90] American Telephone and Telegraph (AT&T). *UNIX System V Release 4: Software Technology Overview*, June 1990.

[AR89] Vadim Abrossimov and Marc Rozier. Generic Virtual Memory Management for Operating System Kernels. *Proceedings of the 12th ACM Symposium on Operating System Principles*, 23(5):123–136, December 1989. Chorus Systems.

[Bac86] Maurice J. Bach. *The Design Of The Unix Operating System*. Prentice Hall, 1986.

[BBB+87] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Wayne Young. *MACH Kernel Interface Manual*. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, January 1987.

[Bel66] L. A. Belady. A Study of Replacement Algorithms for Virtual Storage Systems. *IBM Systems Journal*, 5(2):78–101, 1966.

[BFS89] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, December 1989.

[BMK88] David R. Boggs, Jeffrey C. Mogul, and Christopher A. Kent. Measured Capacity of an Ethernet: Myths and Reality. In *SIGCOMM '88: Symposium on Communications, Architectures, and Protocols*, pages 222–234. ACM SIGCOMM, August 1988.

[BN84] A.D. Birrel and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[CCG+91] Steve J. Chapin, Douglas Comer, James Griffioen, Scott Mark, Patrick Muckelbauer, and Shawn Ostermann. Porting an X11R4 Server to the Xinu Operating System. In *Proceedings of the Xhibition 91 Conference*, pages 131–139, June 1991.

[CG90a] Douglas Comer and James Griffioen. A New Design for Distributed Systems: The Remote Memory Model. In *Proceedings of the Summer 1990 USENIX Conference*, pages 127–135. USENIX Association, June 1990.

[CG90b] Douglas Comer and James Griffioen. Cooperative Management of Embedded Resources in a Distributed Environment. Technical Report CSD-TR-1034, Department of Computer Science, Purdue University, October 1990.

[CG91] Douglas Comer and James Griffioen. Virtual Memory Xinu. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems.* USENIX Association, March 1991. Also released as Technical Report No. CSD-TR-1028, Department of Computer Science, Purdue University.

[Che86] David R. Cheriton. VMTP: A Transport Protocol for the Next Generation of Communication Systems. In *SIGCOMM '86: Symposium*, pages 406–415. ACM, August 1986.

[Che88] David Cheriton. VMTP: Versatile Message Transaction Protocol. ARPANET Working Group Requests For Comments, Febuary 1988. RFC 1045.

[CLZ87] D.D. Clark, M.L. Lambert, and L. Zhang. NETBLT:A High Throughput Transport Protocol. In *SIGCOMM '87 Workshop*, pages 353–359. ACM, August 1987.

[CM88] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[CMDD62] F. J. Corbato, M. Merwin-Daggett, and R. C. Daley. An Experimental Time-Sharing System. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 335–344, 1962.

[Com84] Douglas Comer. *Operating System Design: The XINU Approach.* Prentice–Hall, 1984.

[Com87] Douglas Comer. *Operating System Design, Volume II: Internetworking with XINU.* Prentice–Hall, 1987.

[Com88] Douglas Comer. *Internetworking With TCP/IP: Principles, Protocols, and Architecture.* Prentice Hall, Inc., 1988.

[CS91] Douglas E. Comer and David L. Stevens. *Internetworking With TCP/IP, Volume II: Design, Implementation, and Internals.* Prentice Hall, Inc., 1991.

[Den70] Peter J. Denning. Virtual Memory. *Computing Surveys*, 2:153–189, September 1970.

[Den80] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6:64–84, January 1980.

[Dew88] Prasun Dewan. Supporting Objects in a Conventional Operating System. Technical Report CSD-TR-762, Computer Sciences Department, Purdue University, April 1988.

[Dig80] Digital Equipment Corporation, Intel, Xerox. *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications (Version 1.0)*, 1980.

[Dig84] Digital Equipment Corporation. *QMA DMV11 Synchronous Controller User's Guide*, 1st edition, January 1984.

[Dig85] Digital Equipment Corporation, Maynard, MA. *VAX Architecture Reference Manual*, 1985.

[Dij68] E.W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[DJA88] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. The Clouds Distributed Operating System. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 2–9. IEEE, June 1988.

[Fin88] Raphael A. Finkel. *An Operating Systems VADE MECUM.* Prentice Hall, 1988.

[FP89] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–222, December 1989.

[Gla89] L. Brett Glass. Disk Caching. *Byte*, 14(10):297–301, October 1989.

[GMS88] Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual Memory Architecture in SunOS, 1988.

[Gri89] James Griffioen. A Virtual Memory Operating System for a Distributed Workstation Environment. Technical Report CSD-TR-884, Department of Computer Science, Purdue University, April 1989.

[Gro85] C. P. Grossman. Cache-DASD Storage Design for Improving System Performance. *IBM Systems Journal*, 24(3/4):316–334, 1985.

[Gro89] C. P. Grossman. Evolution of the DASD Storage Control. *IBM Systems Journal*, 28(2):196–226, 1989.

[Hab89] Lynn Haber. An Inside Track on High-capacity Disk Drives. *Sun Expert*, 1(2):37-48, December 1989.

[Hag89] Robert Hagmann. Comments on Workstation Operating Systems and Virtual Memory. In *Proceedings of the Workshop on Workstation Operating Systems*, September 1989.

[HS82] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures.* Computer Science Press, 1982.

[IBM90a] IBM Corporation. *IBM RISC System 6000 Technology*, first edition, 1990.

[IBM90b] IBM Corporation Advanced Workstation Division, 11400 Burnet Rd, Austin, TX 78758. *POWER Processor Architecture: Version 1.52*, February 1990.

[Kan89] Gerry Kane. *MIPS RISC Architecture.* Prentice Hall, 1989.

[Kei90] Bruce E. Keith. Perspectives on NFS File Server Performance Characterization. In *Proceedings of the Summer 1990 USENIX Conference*, pages 267–277, June 1990.

[Knu69] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison Wesley Publishing Company, 1969.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison Wesley Publishing Company, 1973.

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

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, Department of Computer Science, September 1986.

[Lis72] B. H. Liskov. The Design of the Venus Operating System. *Communications of the ACM*, 15(3):144–149, March 1972.

[LKKQ89] Samuel J. Leffler, Marshal K. Mc Kusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD Unix Operating System.* Addison Wesley, 1989.

[LL82] Henry Levy and Peter Lipman. Virtual Memory Management in the VAX/VMS Operating System. *Computer*, pages 35–41, March 1982. Publication of the IEEE.

[LLD+83] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications*, 1(5):842–856, November 1983.

[LN78] H. C. Lauer and R. M. Needham. On The Duality of Operating System Structures. In *In Proceedings of the Second International Symposium on Operating Systems.* ACM, October 1978. Reprinted in Operating Systems Review, 13, 2 April 1979, pp.3-19.

[LS89] Bob Lyon and Russel Sandberg. Breaking Through the NFS Performance Barrier. *SunTech Journal*, 2(4):21, August 1989.

[Lyo84] Bob Lyon. Sun Remote Procedure Call Specification. Technical report, Sun Microsystems, Inc., 1984.

[LZCZ86] Edward D. Lazowska, John Zahorjan, David R. Cheriton, and Willy Zwaenepoel. File Access Performance of Diskless Workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, August 1986.

[McF76] J.H. McFadyen. Systems Network Architecture: An Overview. *IBM Systems Journal*, 15:2–23, 1976.

[Moc87] P. Mockapetris. Domain Names - Concepts and Facilities, November 1987. RFC 1034.

[Mot89] Motorola. *MC68020 32-Bit Microprocessor User's Manual: Third Edition.* Prentice Hall, 1989.

[Nar88] T. Narten. *Best Effort Delivery in Connectionless Networks.* PhD thesis, Department of Computer Science, Purdue University, West Lafayette, IN, August 1988.

[Nel86] Michael N. Nelson. Virtual Memory for the Sprite Operating System. Technical Report UCB/CSD 83/301n, University of California Berkeley, June 1986.

[Nye90] Adrian Nye. *X Protocol Reference Manual.* O'Reilly and Associates, Inc., 1990.

[OCD+87] John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch. The Sprite Network Operating System. Technical Report UCB/CSD 87/359n, University of California Berkeley, June 1987.

[Org72] E. I. Organick. *The Multics System: An Examination of its Structure.* MIT Press, Cambridge, MA, 1972.

[Org86] International Standards Organization. *Information Processing Systems - Open Systems Interconnection - Connection Oriented Transport Protocol Specification.* ISO, Switzerland, July 1986.

[Ous90] John K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[Par90] C. Partridge. A Faster Data Delivery. *Unix Review*, 8(3):43–48, March 1990.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD 88*, pages 109–116, June 1988.

[Pos80a] J. Postel. DOD Standard Internet Protocol, January 1980. RFC 760.

[Pos80b] J. Postel. User Datagram Protocol, August 1980. RFC 768.

[Pos81] J. Postel. Transmission Control Protocol—DARPA Internet program protocol specification, September 1981. RFC 793.

[PPTT90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. Technical report, AT&T Bell Labs, 1990.

[Pre91] Larry Press. Personal Computing - Notes from COMDEX: The Plus Hardcard. *Communications Of The ACM*, 34(7):28, July 1991.

[PS85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts*. Addison-Wesley Publishing Company, 1985.

[QSP85] J. S. Quarterman, A. Silberschatz, and J. L. Peterson. 4.2BSD and 4.3BSD as Examples of the UNIX System. *Computing Surveys*, 17(4), December 1985.

[Ras86] Rick Rashid. Threads Of A New System. *Unix Review*, 4:37–49, August 1986.

[RHF90] Floyd E. Ross, James R. Hamstra, and Robert L. Fink. FDDI - A LAN Among MANs. *Computer Communication Review*, pages 16–31, 1990.

[RK88] Umakishore Ramachandran and M. Yousef A. Khalidi. An Implementation of Distributed Shared Memory. Technical Report GIT-ICS-88/50, School of Information and Computer Science, Georgia Institute of Technology, December 1988.

[RST89] Robbert Van Renesse, Hans Van Staveren, and Andrew S. Tanenbaum. The Performance of the Amoeba Distributed Operating System. *Software: Practice and Experience*, 19(3):223–234, March 1989.

[RT74] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[Rus91] Vincent F. Russo. *An Object-Oriented Operating System*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[SAG+72] K.C. Sevick, J. W. Atwood, M. S. Grushcow, R. C. Holt, J. J. Horning, and D. Tsichritzis. Project SUE as a Learning Experience. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 331–337, 1972.

[San85] Russel Sandberg. Sun Network Filesystem Protocol Specification. Technical report, Sun Microsystems, Inc., 1985.

[SG86] Robert W. Scheifler and Jim Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer USENIX Conference*, pages 119–130. USENIX Association, June 1985.

[Sol81] K. R. Sollins. The TFTP Protocol: Revision 2, June 1981. RFC 783.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments In System Design. *ACM Transactions on Computer Systems*, 2:277–288, November 1984.

[Sta91] William Stallings. *Data and Computer Communications.* Macmillan Publishing Company, 1991. Third Edition.

[Sun] Sun Microsystems, Inc. *The NeWS Programmer's Guide.*

[Sun86] Sun Microsystems, Inc. *Writing Device Drivers for the Sun Workstation*, 1986.

[Sun87] Sun Microsystems, Inc. *The SPARC Architecture Manual: Version 7*, October 1987.

[Sun90] Sun Microsystems, Inc. *Release Report: SunOS 4.1*, January 1990.

[Tan81] A. Tannenbaum. *Computer Networks.* Prentice Hall, Inc., 1981.

[Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall, 1987.

[Tev87] Avadis Tevanian. Architecture Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach. Technical Report CMU-CS-88-106n, CMU, December 1987. PhD Thesis.

[TvRvS$^+$90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba Distribute Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[Wel86] Brent B. Welch. The Sprite Remote Procedure Call System. Technical Report UCB/CSD 86/302n, University of California Berkeley, June 1986.

[Wil89] David Wilson. Tested Mettle. *Unix Review*, 7(8), August 1989.

[YTR87] Michael Young, Avadis Tevanian, and Richard Rashid. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the ACM Symposium on Operating System Principles.* ACM, November 1987.