1990

A New Design for Distributed Systems: The Remote Memory Model

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

James Griffioen

Report Number:
90-977

A NEW DESIGN FOR DISTRIBUTED SYSTEMS: THE REMOTE MEMORY MODEL

Douglas Comer
James Griffioen

CSD-TR-977
April 1990

A New Design for Distributed Systems: The Remote Memory Model

Douglas Comer
comer@cs.purdue.edu
(317) 494-6009

James Griffioen
jng@cs.purdue.edu
(317) 494-7836

Department of Computer Science
Purdue University
West Lafayette, IN
47907
CSD-TR-977

ABSTRACT

This paper describes a new model for constructing distributed systems called the *Remote Memory Model*. The remote memory model consists of several client machines, one or more dedicated machines called *remote memory servers*, and a communication channel interconnecting them. In the remote memory model, client machines share the memory resources located on the remote memory server. Client machines that exhaust their local memory move portions of their address space to the remote memory server and retrieve pieces as needed. Because the remote memory server uses a machine-independent protocol to communicate with client machines, the remote memory server can support multiple heterogeneous client machines simultaneously.

This paper describes the remote memory model and discusses the advantages and issues of systems that use this model. It examines the design of a highly efficient, reliable, machine-independent protocol used by the remote memory server to communicate with the client machines. It also outlines the algorithms and data structures employed by the remote memory server to efficiently locate the data stored on the server. Finally, it presents measurements of a prototype implementation that clearly demonstrate the viability and competitive performance of the remote memory model.

1. Background

Virtual memory provides a necessary abstraction for writing architecture-independent portable programs. In a virtual memory architecture, each program has a large, linear address space in which it places code and data. Applications may use more memory than physically available, freeing the programmer from the responsibility of physical memory management. The operating system creates the virtual memory illusion using secondary memory for backing storage when the system exhausts physical memory.

Many conventional virtual memory systems use random access magnetic disks for backing storage[8]. High data transfer rates, random access capabilities, and large capacity make disks a desirable form of backing storage. Many virtual memory systems use demand paging to retrieve data from secondary storage on demand when the system accesses the data. The desire to service page faults quickly makes a disk's fast random access capability extremely valuable. In addition, most operating systems use disks both for backing storage and file storage.

Because disks have become such a prominent form of backing storage, we almost always associate backing storage with disks. Even distributed systems containing diskless machines use remote disks for backing storage. Instead of connecting a disk to each workstation, many distributed systems allow diskless clients to share a disk resource over the network through a special file server machine. Sun Microsystem's SunOS running on a diskless workstation accesses a remote NFS file on disk for backing storage[1,10]. Similarly, the Sprite operating system uses a file on its remote file system for virtual memory backing storage[7,12]. The Mach and Chorus operating systems allow users to define their own backing store and paging methods[3,11], however, most implementations still use a local disk for backing storage.

Equating disks with backing storage impacts the way we design virtual memory operating systems. Operating systems use several techniques to minimize the time spent moving memory to and from backing storage. To optimize head movement on the disk, the operating system often groups write operations together and issues them all at once. To avoid extra bus traffic and disk activity, the operating system delays moving memory to backing storage until absolutely necessary.

Clearly, technology has significantly impacted the way we design systems. This paper presents a new model for designing distributed systems based on remote memory backing storage. The paper describes the impact remote memory has on the design and functionality of such a system and presents a prototype implementation along with experimental results.

2. The Remote Memory Model

The *remote memory model* provides a new basis for designing distributed systems. The model consists of several client machines, various server machines, one or more dedicated machines called *remote memory servers*, and a communication channel interconnecting all the machines. Figure 1 illustrates one possible remote memory model architecture. In the remote memory model, client machines use remote memory for backing storage rather than disks. The remote memory server provides a shared resource for all client machines. Clients use the communication channel to access memory on the remote memory

server.

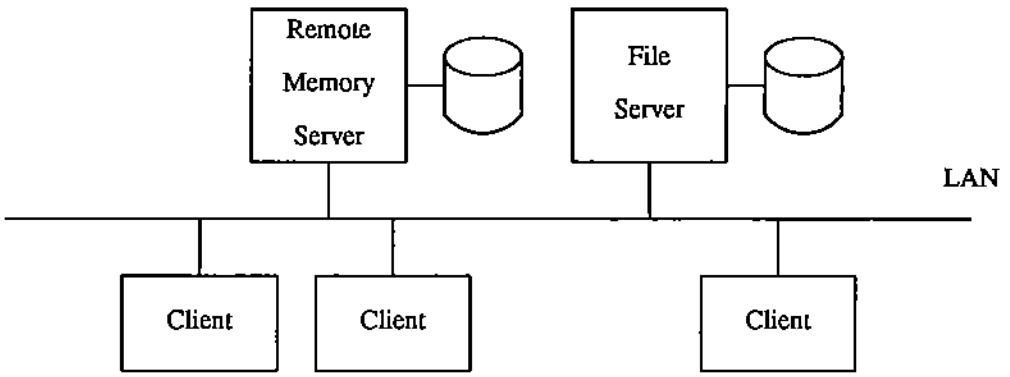

Figure 1: An Example Remote Memory Model Architecture

Figure 1 pictures several diskless client machines connected to a local area network (LAN). The diskless client machines access a remote file server for file storage. Client machines may use a local disk or a remote file server for file storage, but, in either case, the client machines use the remote memory server for backing storage. Each client's operating system supports multiple users in separate virtual address spaces. The operating system also provides the functionality needed to access remote memory.

Each client machine has a private local memory large enough to support the usual processing demands of the client. The remote memory server provides additional memory for applications requiring exceptionally large amounts of memory. When the client exhausts the local physical memory, the operating system moves data to the remote memory server, and retrieves the data as needed.

To keep the model as general as possible, we assume the system consists of heterogeneous machines. Each remote memory server provides remote memory backing storage for multiple heterogeneous client machines simultaneously. The system may contain several remote memory servers. Although the model, in its most general form, does not prohibit a client from communicating with more than one remote memory server, we simplify the model by assuming that each client machine only accesses one predefined remote memory server. Each remote memory server provides a large physical memory resource used to store client data. Remote memory servers may use a local disk to provide additional storage capacity, if needed.

The communication channel allows client machines to send and receive data to and from the remote memory server and other servers. We assume the communication channel has a relatively high bandwidth and low delay (for example, an 10Mb/s Ethernet). To encompass as many communication technologies as possible, we assume the communication channel provides unreliable packet delivery.

The remote memory server makes the remote memory model uniquely different from conventional distributed systems. In conventional distributed systems, each client's virtual memory system is completely independent from all of the other client's virtual memory systems. Even though clients access a common disk via a file server, their virtual memory systems do not interact in any way. The remote memory model takes a different approach, viewing the memory on the remote memory server as a part of the total memory available to the system. In the remote memory model, the remote memory server dynamically allocates regions of its memory to clients that require additional memory space, continually reassigning portions of its memory to clients according to their needs. Clients share the memory space available on the server with all other client machines. As a result, the remote memory server expands the memory available to each client machine. This difference from conventional system gives the remote memory model several desirable properties.

Additional Memory: Because clients share a large remote memory resource, client machines may obtain additional memory for applications that require large amounts of memory. We assume clients have enough local memory to support the usual processing demands, but request additional memory space from the remote memory server for large applications.

Arbitrarily Large Storage Capacity: The memory server may employ a form of virtual memory to present an arbitrarily large memory resource to the client machines. The server may connect to one or more disk drives and use a memory replacement scheme to substantially enlarge the storage capacity of the server. As a result, the server presents a large memory space to clients even though the server has a finite physical memory capacity. When the clients exhaust the physical memory on the server, the server moves some of the client's data to disk, making room for more client data in physical memory. Client machines do not know the size of the physical memory on the remote memory server and do not need to restrict application memory usage in any way. From the client's viewpoint, the server simply provides a large memory resource.

Data Sharing: The remote memory server provides a centralized memory, allowing homogeneous client machines to efficiently share data. Although the remote memory model permits data sharing between clients via the remote memory server, the model does not specify the mechanism used to share data. Depending on the type of data sharing desired, the remote memory server may implement mechanisms that allow read only sharing, read-write sharing, or no sharing[6]. Because homogeneous client machines often execute the same applications, use the same shared libraries, and execute the same kernel code, even simple data sharing mechanisms can significantly reduce the amount of server memory used by client machines.

Offloading File Server: The remote memory model improves file system performance by removing paging activity from the file system. Removing paging activity from the file system significantly reduces contention for the disk and eliminates many extra head movement operations. Paging activity tends to access data regions in a random fashion, while file activity tends to access data sequentially. Because the nature of paging activity differs from the nature of file activity, separating paging activity from file activity allows us to implement each operation efficiently. Unlike a remote file server, the remote memory server understands paging activity and makes intelligent decisions regarding storage and retrieval of the data.

Remote Memory Semantics: The remote memory model allows us to define the remote memory semantics in a way that meets the reliability requirements of the system as a whole. For example, we may view the memory on the remote memory server as one component of the total system memory. If any part of the system's memory fails to operate correctly, such as the client's local memory or the remote memory server's memory, the system may lose valuable data. Given these semantics, we do not require the remote memory server to maintain multiple or permanent copies of the data. A different definition of the remote memory semantics may require the server to provide reliable storage. In this case, the remote memory server must maintain duplicate copies of all data, either on other servers or on a permanent storage device such as a disk. Still another definition may require remote memory to provide reliable storage and reliable retrieval. In this case, multiple servers may cooperate to insure that clients can access remote memory at all times.

3. The Design

The prototype design concentrates on three components of the system: the client virtual memory operating system, the remote memory server, and the protocol clients and servers use to communicate. The design goals we wanted to achieve were:

- to design a communication protocol independent of the underlying network architecture.
- to design a communication protocol that guarantees reliable delivery.
- to make the remote memory resource available to heterogeneous client architectures and heterogeneous client operating systems simultaneously.
- to create a system that provides efficient interaction between client and server without sacrificing the generality mentioned in the previous design goals.

As a result, we designed a protocol that provides reliability, architecture independence, and efficiency, along with a remote memory server capable of efficiently supporting multiple heterogeneous client

architectures simultaneously.

3.1. The Remote Memory Communication Protocol

The remote memory communication protocol used between clients and a remote memory server consists of two layers: the Xinu Paging Protocol (XPP) layer, and the Negative Acknowledgement Fragmentation Protocol (NAFP) layer.

3.1.1. The XPP Layer

Client operating systems use XPP to reliably transfer memory to and from the remote memory server. XPP supports four basic message types: page store requests, page fetch requests, process create requests, and process terminate requests. When a client machine exceeds the capacity of the local physical memory, the client issues a page store request to store data on the remote memory server. Later, the client issues a page fetch request to retrieve the data from the server. When a client creates or terminates a process, it informs the remote memory server using an create request or a terminate request which we describe in greater detail in section 3.2.3. All XPP request messages originate at the client. The server accepts XPP messages, processes them, and sends reply messages.

XPP provides a communication mechanism independent of the client architectures comprising the remote memory model, permitting heterogeneous client machines to communicate with a single remote memory server. Because the virtual memory system on each client has its own page size, XPP page store and page fetch requests allow clients to transfer variable size memory regions to and from the server. XPP transfers pages of any size, regardless of the underlying communication channel's transport characteristics or maximum packet size.

The paging activity between a client and a server requires reliable, in-order, deliver of all messages. If XPP did not reliably deliver messages in-order, a store request followed by a fetch request could arrive out of order, or a store request could be lost, causing incorrect results. XPP employs sequence numbers, positive acknowledgement, timeouts, and retransmissions to insure reliable, in-order, processing of XPP messages.

Each client machine assigns a sequence number to every XPP message it sends. The remote memory server remembers the sequence number of the last XPP message received from each client machine. The sequence number serves two purposes: it uniquely identifies each message and imposes an ordering on the list of messages. Although the sequence numbers define a processing order that insures correctness, the system does not need to impose such a strict ordering on message processing to achieve

correct results. For example, the server may process a message for page i before or after a message for page j and still obtain correct results. However, if a client sends a store request for page i followed by a fetch request for page i, the server must process the store request before the fetch request. XPP achieves correctness by defining a partial ordering on the list of messages. In addition to a sequence number, each XPP message contains a *preceeding message number*. The preceeding message number specifies the sequence number of the most recent preceeding message that must be processed before the current message can be processed. The server may process any message as long as it has already processed the associated preceeding message.

XPP uses end-to-end positive acknowledgements (ACKs) to indicate that the server performed the requested operation[9]. XPP does not use acknowledgements to indicate that a message was successfully transmitted like many other protocols do. Instead, an XPP ACK provides an end-to-end acknowledgement, signaling the successful completion of the requested high level operation. When the server completes the requested operation, it sends a positive acknowledgement to the client containing the results of the operation. Client machines do not discard local copies of pages until the server acknowledges that it has stored the data in remote memory. Most XPP ACKs simply indicate success or failure; however, page fetch replies contain the requested data in addition to the status field.

XPP guarantees reliable delivery using timeouts and retransmissions. The client machine timestamps each message sent to the server, and then holds the message until the server replies with an XPP ACK. If the client does not receive an ACK within a predefined timeout period, the client resends the message. Because all messages originate on the client side, the server never initiates a request and does not need to implement the timeout/retransmission mechanism, greatly simplifying the implementation and improving efficiency.

3.1.2. The NAFP Layer

The Negative Acknowledgement Fragmentation Protocol (NAFP) provides the support needed to efficiently transport XPP messages over a wide variety of communication channels. To allow operation over as many network architectures as possible, the remote memory communication protocol assumes, as a minimal requirement, that the communication channel provides unreliable datagram service.

Because XPP supports a wide variety of page sizes, the length of an XPP message may exceed the maximum packet size of the underlying physical communication channel. NAFP accepts an entire XPP message and breaks the message into fragments. NAFP then transmits each fragment over the communication channel, reassembles the fragments into a complete message, and presents the message to the

receiving XPP layer. Because NAFP and XPP adhere to the network layering principle, the sending and receiving XPP layers see exactly the same message[4].

Although conventional communication channels, for instance local area networks, provide reasonably reliable packet delivery, they still drop packets, deliver packets out of order, deliver packets late, or corrupt packet contents. The XPP layer corrects such errors and guarantees reliable delivery. However, the XPP reliability mechanism usually detects communication errors long after the error occurs. XPP then pays a high cost to correct the error. For example, to send or receive an 8K byte page from a Sun3/50 over an Ethernet with a maximum transmission unit (MTU) of approximately 1500 bytes requires a minimum of 6 packets. If the communication channel loses or corrupts any 1 of the 6 packets, XPP will not detect the error until the timeout occurs, and then it must resend all 6 fragments. Because XPP guarantees reliable delivery, NAFP does not need to correct any errors. However, to improve efficiency, NAFP attempts to detect and correct errors as soon as they happen, improving, but not guaranteeing, reliability.

NAFP improves reliability using negative acknowledgements (NACKs). NAFP assigns a sequence number to each fragment and sends each fragment in order. As packets arrive on the receiving end, the NAFP layer reassembles the XPP message but does not acknowledge any of the fragments. As long as no communication errors occur, NAFP transfers messages efficiently with no additional overhead. A fragmentation error arises when a fragment arrives out of order. The receiving end remembers the sequence number of the last fragment for each partially transmitted message. As soon as the NAFP layer receives a fragment out of order, the receiving NAFP layer sends a negative acknowledgement to the sending NAFP layer containing the missing fragment's sequence number. The sending NAFP layer receives the NACK and resends the missing fragment. NAFP does not guarantee reliable delivery of fragments. Instead, NAFP makes a half-hearted attempt to correct errors, sending a single NACK for each missing fragment in hope that the sender will receive the NACK and resend the missing fragment. If the simple, low cost, NAFP error correction mechanism fails, XPP will detect the error and take the corrective measures needed to reliably deliver the message. Because NAFP improves reliability, XPP rarely retransmits messages. In the expected case, in which no communication errors occur, NAFP incurs no additional overhead.

Because the remote memory communication protocol imposes minimal requirements on the communication channel, we can use the remote memory communication protocol over any transport mechanism that provides unreliable datagram service. Almost any protocol can function as the underlying communication protocol, including reliable datagram protocols, stream protocols, or virtual circuit protocols.

To achieve independence from the underlying physical network architectures, we use an architecture-independent protocol like UDP or VMTP as the underlying datagram protocol. Architecture independent protocols like UDP and VMTP allow the remote memory communication protocol to operate over local area networks comprised of several different physical network architectures.

3.2. The Remote Memory Server

Client performance depends, to a large extent, on the delay the network and the remote memory server introduce. To improve client performance, the remote memory server attempts to minimize the delay by using the remote memory communication protocol and efficient data look-up algorithms. To support as many client machines as possible, the remote memory server avoids preallocation of resources in order to make efficient use of memory.

One of our goals required support for heterogeneous client access to the remote memory resource. Using the remote memory communication protocol, the remote memory server transfers data to and from heterogeneous clients in an architecture-independent manner. The remote memory server supports all architectures regardless of byte size or byte order. Although the remote memory server maintains information about each memory segment it stores, the server does not attempt to interpret or modify the stored data. The server simply returns data in exactly the same form in which it was received.

3.2.1. Efficient Use of Memory

Heterogeneous machines running heterogeneous operating systems use a wide variety of page sizes. The remote memory server uses dynamically allocated data structures to store the variable size memory segments clients send to the server. The remote memory server divides the available memory into small fixed size segments or data blocks. When a client machine sends a page store request to the remote memory server, the server allocates the precise number of data blocks needed to store the page. The tradeoff between data block management overhead and memory space utilization makes it difficult to chose an optimal data block size. Using large data blocks causes internal memory fragmentation, and using small data blocks increases the data block 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 reasonable. In practice, only a few popular page sizes exist based on powers of two, making the choice easy.

The remote memory server allocates data blocks dynamically for each new store request. The data structures do 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 data, the server may spread the data across several disjoint data blocks. The ability to scatter the data from a store request across memory results in efficient use of memory and provides support for a wide variety of page sizes. Because many of the client machines have the same page size, the remote memory server usually allocate and frees segments of the same size, resulting in decreased external memory fragmentation. Consequently, the server can usually allocate contiguous data blocks to each new store request.

3.2.2. Efficient Data Look-Up

To reduce the delay associated with retrieving memory from the remote memory server, the server attempts to minimize the time spent searching the data structures for the desired data. The server uses a *data* hash table and a double hashing algorithm to locate data. Data hash table entries maintain information about client pages stored on the server. Each active hash table entry contains information for exactly one client page, including information regarding the identity of the page, and a range or list of data blocks containing the data. The remote memory server uses a single *data* hash table to store all the data from all the client machines. Client machines uniquely identify a page with an ordered triple consisting of a unique machine identifier, a process identifier, and a page identifier. The server applies a double hashing algorithm to the triple to locate the hash table entry that contains pointers to the data[5]. If the hash table is less than 95% full, the double hashing algorithm, on the average, locates a saved page in less than three probes to the table. As long as the remote memory server limits the utilization of the hash table to less than 95% of the total capacity, the average look-up time remains constant, regardless of the amount of data the clients store on the server or the number of clients using the server.

3.2.3. Memory Reclamation

The remote memory server employs an efficient memory deallocation algorithm to amortize the cost of reclaiming memory over time. The algorithm allows clients to free large amounts of memory with a single inexpensive operation. We assume that most client operating systems support multiple processes and create and terminate processes frequently. To make process termination efficient, client machines require the ability to free large amounts of remote memory in a single operation.

The XPP protocol does not provide a message for releasing individual pages on the server. Instead, XPP provides a terminate process request message. When a process exits, the operating system issues an XPP terminate process request message. The responsibility for freeing all the remote memory associated with the process falls on the remote memory server.

To avoid spending large amounts of time searching for pages associated with the terminated process, the remote memory server maintains a second *process* hash table containing information about all active processes on all client machines. The server maintains a timestamp for each process in the system. When the server receives a page store request for a process, the server saves the process's timestamp with the page in the *data* hash table. Each time the server receives a terminate request, the server updates the timestamp in the process hash table, thereby invalidating all pages associated with the terminated process. The server reclaims obsolete pages during later probes to the data hash table and with a garbage collection process that executes in the background. Each time a probe to the data hash table results in a collision, the server checks the timestamp on the page against the timestamp of the owner. If the timestamps differ, the server reclaims the page. Together, the garbage collection process and the lazy reclamation algorithm amortize the cost of reclaiming memory over time.

4. A Prototype Implementation and Experimental Results

We designed and implemented a prototype remote memory distributed system based on the remote memory model. The system consists of heterogeneous client machines (Sun Microsystems Sun 3/50's, Digital Equipment Corporation Microvax I's and II's), a remote memory server machine (we have used a Sun 3/50, Vax 11/780, Microvax III, Vaxserver 3100, and an 8 processor Sequent Symmetry as a remote memory server), a file server machine (a Vax 11/780 or Sun 3/50), all connected by a 10 Mb/sec Ethernet. Sun and Microvax client machines simultaneously access the remote memory server for backing storage, demonstrating support for heterogeneous clients.

In the prototype, remote memory is high speed volatile storage, susceptible to failure and data loss. To keep the prototype simple, the remote memory server does not support any data sharing between client machines. After experience with the server, we chose 1K byte blocks as the storage page size.

We built the remote memory communication protocol on top of UDP to allow communication over almost any network architecture. Moreover, UDP allows client machines to reside on a different physical network than the server machine. We have experimented with a configuration in which client machines access a remote memory server on a remote network through several gateways. Even when traversing several gateways to access the remote memory server, the high-cost XPP guaranteed reliability mechanism rarely retransmits messages because the negative acknowledgement fragmentation protocol corrects most communication errors. Another configuration we have used chains remote memory servers together. Client machines access a diskless remote memory server that in turn accesses a remote memory server with a disk.

Our initial timing results show that storing or retrieving an 8K byte segment between a Sun 3/50 client and a Sun 3/50 remote memory server requires an average of 39ms. In contrast, current production systems consisting of diskless Sun 3/50s paging over NFS require an average of 50ms to process an 8K byte read request when accessing a file sequentially. To randomly access an NFS file, as paging activity does, requires an average of 84ms to process an 8K byte read request and an average of 176ms to process an 8K byte write request.

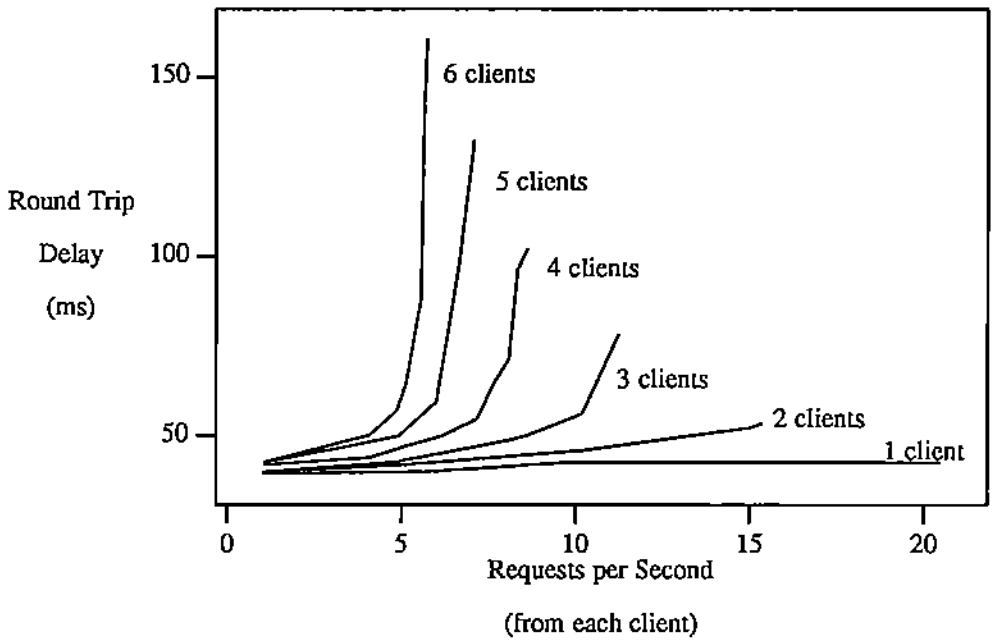

Figure 2: Round Trip Delay As A Function Of The Client Request Rate

Figure 2 shows the cost of performing a page fetch operation (measured as round trip delay) as a function of the number of requests issued per second by each client machine. We conducted the tests using Sun 3/50 client machines paging to a Sun 3/50 remote memory server and implemented the prototype remote memory server as a UNIX application level process. Because the Ethernet has an MTU of 1500 bytes, the NAFP protocol breaks each 8K byte Sun 3/50 page into 6 Ethernet packets. All client machines send paging requests concurrently. Each client sends requests at a constant rate, uniformly distributed over time, to the remote memory server. The request rates shown in the figure indicate the number of requests per second issued by a single client machine.

The figure shows the round trip delay for a varying number of clients and request rates. The sudden rise in the round trip delay shown in the curves for 3 or more clients can be misleading. The following figure shows that the sudden increase in each curve occurs at the point where the server becomes

overloaded (total load of 30 requests/second).

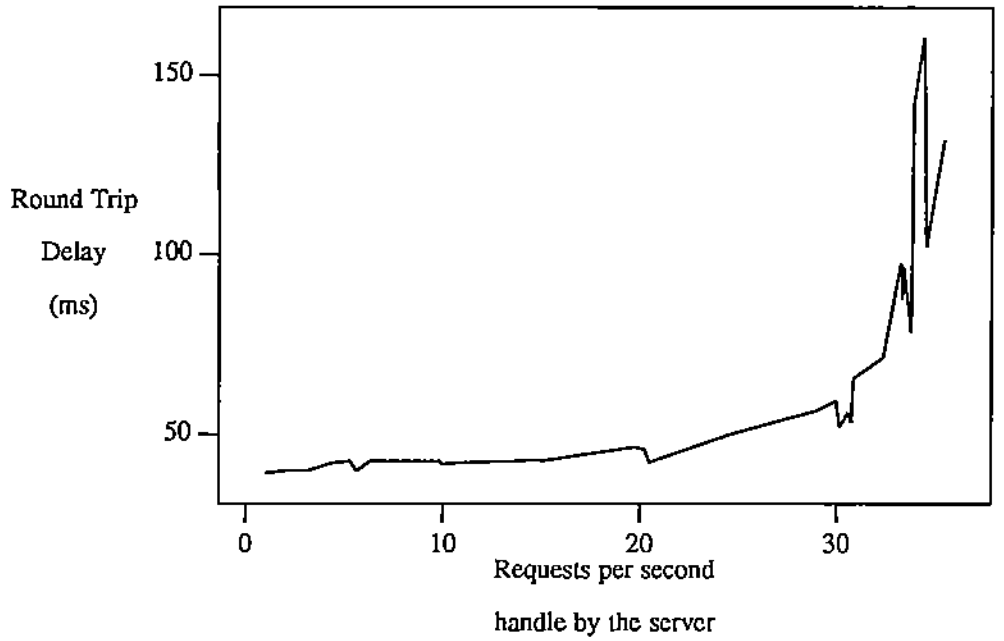

Figure 3: Remote Memory Delay For Various Server Loads

Figure 3 illustrates the average round trip delay as a function of the number of requests the server processes per second. We generate the server load by varying the number of clients and the rate at which they send requests to the server. The figure shows that the current prototype remote memory server, executing as a user level process on a Sun 3/50, efficiently handles up to 30 requests per second. At 30 requests per second the prototype memory server becomes saturated and any more load on the server significantly increases the round trip delay, explaining the sharp rise in the curves pictured in figure 2. For loads of less than 30 requests per second, the average round trip delay remains less than 56ms regardless of the number of clients, If the load on the server is less than 20 requests per second (2/3 of the server's capacity), the round trip times never exceeds 46ms.

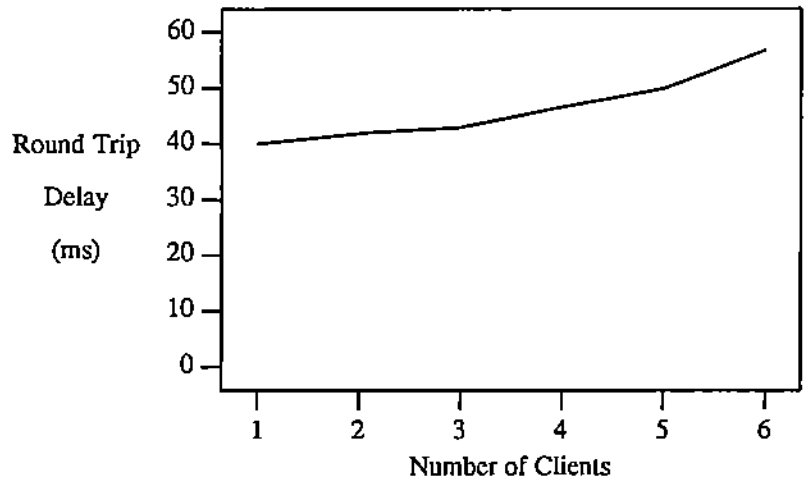

Figure 4: Round Trip Time (Each Client Sends 5 Requests/Second)

If we assume, under usual operating circumstances, that clients send an average of 5 requests per second, then figure 4 illustrates the gradual increase in round trip delay as the number of clients increases to the maximum capacity of the server. As the number of client increases, the number of dropped fragments increases, resulting in slightly higher round trip times. Lyon and Sandberg indicate that 8 diskless Sun 3/50 workstations generate an average load of 30 NFS requests per second, or an average of 3.75 requests per second per client, which includes both file activity and paging activity[2]. Consequently, an average of 5 paging requests per second per client may be somewhat high, indicating that the slope of the line in figure 4 would be even less for the usual workload.

5. Conclusions

Using the remote memory model as an alternative model for designing distributed systems has many attractive properties. The large memory resource shared by all client machines is especially appealing. Experience with the prototype system clearly demonstrates the viability of the remote memory model and shows that performance is competitive with distributed systems currently in use. Finally, we showed that the remote memory model can support heterogeneous clients machines without sacrificing efficiency.

References

1. R. Sandberg , D. Goldberg, S. Kleiman, Dan Walsh, and Bob Lyon, "Design and Implementation of the Sun Network File System," Proceedings of the Summer USENIX Conference, pp. 119-130, USENIX Association, June 1985.

2. R. Sandberg and Bob Lyon, "Breaking Through the NFS Performance Barrier," SunTech Journal, p. 21, August 1989.

3. Vadim Abrossimov and Marc Rozier, "Generic Virtual Memory Management for Operating System Kernels," *Proceeding of the 12th ACM Symposium on Operating System Principles*, vol. 23, no. 5, pp. 123-136, Chorus Systems, December 1989.

4. Douglas Comer, *Internetworking with TCP/IP: Principles, Protocols, and Architecture*, Prentice Hall, 1988.

5. Donald E. Knuth, *Sorting and Searching*, Addison Wesley Publishing Company, 1973.

6. Kai Li and Paul Hudak, "Memory Coherence in Shared Virtual Memory Systems," Proceedings of the 5th ACM Symposium of Principles of Distributed Computing, pp. 229-239, August 1986.

7. John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch, "The Sprite Network Operating System," Tech Report UCB/CSD 87/359n, University of California Berkeley, June 1987.

8. James L. Peterson and Abraham Silberschatz, *Operating System Concepts*, Addison Wesley, 1985.

9. J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, pp. 277-288, 1984.

10. Robert A. Gingell and Joseph P. Moran and William A. Shannon, *Virtual Memory Architecture in SunOS*, Sun Microsystems, Inc., 1988.

11. Avadis Tevanian, "Architecture Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach," Tech Report CMU-CS-88-106n, CMU, December 1987.

12. Brent B. Welch, "The Sprite Remote Procedure Call System," Tech Report UCB/CSD 86/302, University of California Berkeley, June 1986.

Dr. Douglas Comer is a full professor in the Computer Science Department at Purdue University where he teaches graduate-level courses in operating systems, internetworking, and distributed systems. He has written numerous research papers and five textbooks, and has been principle investigator on many research projects. He designed and implemented the X25NET and Cypress networks, as well as the Xinu operating system. He heads the Xinu, Cypress, Shadow Editing, and Multiswitch research projects. He is a member of the Internet Research Steering Group and chairman of the Internet Naming Research Group. He is a former member of the CSNET Executive Committee and the Internet Activities Board. Professor Comer teaches networking seminars for Interop Incorporated. He is a member of the ACM, AAAS, and Sigma Xi.

James Griffioen is a PhD candidate in the Computer Science Department at Purdue University. His research interests include operating systems and distributed systems. He has worked on the Xinu and the Shadow Editing research projects. He received an MS degree in computer science from Purdue University in 1988 and a BS degree in computer science from Calvin College in 1985. He received the USENIX scholarship for the 89-90 academic year and is a member of the ACM.