The Distributed V Kernel and its Performance for Diskless Workstations

David R. Cheriton and Willy Zwaenepoel

Computer Systems Laboratory
Departments of Computer Science and Electrical Engineering
Stanford University

Abstract

The distributed V kernel is a message-oriented kernel that provides uniform local and network interprocess communication. It is primarily being used in an environment of diskless workstations connected by a high-speed local network to a set of file servers. We describe a performance evaluation of the kernel, with particular emphasis on the cost of network file access. Our results show that over a local network:

1. Diskless workstations can access remote files with minimal performance penalty.
2. The V message facility can be used to access remote files at comparable cost to any well-tuned specialized file access protocol.

We conclude that it is feasible to build a distributed system with all network communication using the V message facility even when most of the network nodes have no secondary storage.

1. Introduction

The distributed V kernel is a message-oriented kernel that provides uniform local and network interprocess communication. The kernel interface is modeled after the Thoth [3, 5] and Verex [4, 5] kernels with some modifications to facilitate efficient local network operation. It is in active use at Stanford and at other research and commercial establishments. The system is implemented on a collection of MC68000-based SUN workstations [2] interconnected by a 3 Mb Ethernet [9] or 10 Mb Ethernet [7]. Network interprocess communication is predominantly used for remote file access since most SUN workstations at Stanford are configured without a local disk.

This paper reports our experience with the implementation and use of the V kernel. Of particular interest are the controversial aspects of our approach, namely:

1. The use of diskless workstations with all secondary storage provided by backend file servers.
2. The use of a general purpose network interprocess communication facility (as opposed to special-purpose file access protocols) and, in particular, the use of a Thoth-like interprocess communication mechanism.

The more conventional approach is to configure workstations with a small local disk, using network-based file servers for archival storage. Diskless workstations, however, have a number of advantages, including:

1. Lower hardware cost per workstation.
2. Simpler maintenance and economies of scale with shared file servers.
3. Little or no memory or processing overhead on the workstation for file system and disk handling.
4. Fewer problems with replication, consistency and distribution of files.

The major disadvantage is the overhead of performing all file access over the network. One might therefore expect that we use a carefully tuned specialized file-access protocol integrated into the transport protocol layer, as done in LOCUS [11]. Instead, our file access is built on top of a general-purpose interprocess communication (IPC) facility that serves as the transport layer. While this approach has the advantage of supporting a variety of different types of network communication, its generality has the potential of introducing a significant performance penalty over the "problem-oriented" approach used in LOCUS.

Furthermore, because sequential file access is so common, it is conventional to use streaming protocols to minimize the effect of network latency on performance. Instead, we adopted a synchronous "request-response" model of message-passing and data transfer which, while simple and efficient to implement as

This work was sponsored in part by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431.

© 1983 ACM 0-89791-115-6/83/010/0129 $00.75

well as relatively easy to use, does not support application-level use of streaming.

These potential problems prompted a performance evaluation of our methods, with particular emphasis on the efficiency of file access. This emphasis on file access distinguishes our work from similar studies [10, 13]. The results of our study strongly support the idea of building a distributed system using diskless workstations connected by a high-speed local network to one or more file servers. Furthermore, we show that remote file access using the V kernel IPC facility is only slightly more expensive than a lower bound imposed by the basic cost of network communication. From this we conclude that relatively little improvement in performance can be achieved using protocols further specialized to file access.

2. V Kernel Interprocess Communication

The basic model provided by the V kernel is that of many small processes communicating by messages. A process is identified by a 32-bit globally unique *process identifier* or *pid*. Communication between processes is provided in the form of short fixed-length messages, each with an associated reply message, plus a data transfer operation for moving larger amounts of data between processes. In particular, all messages are a fixed 32 bytes in length.

The common communication scenario is as follows: A *client* process executes a ***Send*** to a *server* process which then completes execution of a ***Receive*** to receive the message and eventually executes a ***Reply*** to respond with a reply message back to the client. We refer to this sequence as a *message exchange*. The receiver may execute one or more ***MoveTo*** or ***MoveFrom*** data transfer operations between the time the message is received and the time the reply message is sent.

The following sections describe the primitives relevant to this paper. The interested reader is referred to the V kernel manual [6] for a complete description of the kernel facilities.

2.1. Primitives

Send(message, pid)
: Send the 32-byte message specified by *message* to the process specified by *pid*. The sender blocks until the receiver has received the message and has sent back a 32-byte reply using ***Reply***. The reply message overwrites the original message area.

  Using the kernel message format conventions, a process specifies in the message the *segment* of its address space that the message recipient may access and whether the recipient may read or write that segment. A segment is specified by the last two words of a message, giving its start address and its length respectively. Reserved flag bits at the beginning of the message indicate whether a segment is specified and if so, its access permissions.

pid = Receive(message)
: Block the invoking process, if necessary, to receive a 32-byte message in its message vector. Messages are queued in first-come-first-served (FCFS) order until received.

(pid, count) = ReceiveWithSegment(message, segptr, segsize)
: Block the invoking process to receive a message as with ***Receive*** except, if a segment is specified in the message with read access, up to the first *segsize* bytes of the segment may be transferred to the array starting at *segptr*, with *count* specifying the actual number of bytes received.

Reply(message, pid)
: Send a 32-byte reply contained in the message buffer to the specified process providing it is awaiting a reply from the replier. The sending process is readied upon receiving the reply; the replying process does not block.

ReplyWithSegment(message, pid, destptr, segptr, segsize)
: Send a reply message as done by ***Reply*** but also transmit the short segment specified by *segptr* and *segsize* to *destptr* in the destination process' address space.

MoveFrom(srcpid, dest, src, count)
: Copy *count* bytes from the segment starting at *src* in the address space of *srcpid* to the segment starting at *dest* in the active process's space. The *srcpid* must be awaiting reply from the active process and must have provided read access to the segment of memory in its address space using the message conventions described under ***Send***.

MoveTo(destpid, dest, src, count)
: Copy *count* bytes from the segment starting at *src* in the active process's space to the segment starting at *dest* in the address space of the *destpid* process. The *destpid* must be awaiting reply from the active process and must have provided write access to the segment of memory in its address space using the message conventions described under ***Send***.

SetPid(logicalid, pid, scope)
: Associate *pid* with the specified *logicalid* in the specified scope, which is one of local, remote or both. Example *logicalid's* are *fileserver, nameserver,* etc.

pid = GetPid(logicalid, scope)
: Return the process identifier associated with *logicalid* in the specified scope if any, else 0.

2.2. Discussion

The V kernel's interprocess communication is modeled after that of the Thoth and Verex kernels, which have been used in multi-user systems and real-time applications for several years. An extensive discussion of this design and its motivations is available [5], although mainly in the scope of a single machine system. We summarize here the highlights of the discussion.

1. Synchronous request-response message communication makes programming easy because of the similarity to procedure calls.
2. The distinction between small messages and a separate data transfer facility ties in well with a frequently observed usage pattern: A vast amount of interprocess communication is transfer of small amounts of control information (e.g. device completion) while occasionally there is bulk data transfer (e.g. program loading).
3. Finally, synchronous communication and small, fixed-size messages reduce queuing and buffering problems in the kernel. In particular, only small, fixed-size message buffers

must be allocated in the kernel and large amounts of data are transferred directly between users' address spaces without extra copies. Moreover, by virtue of the synchrony of the communication, the kernel's message buffers can be statically allocated. As exemplified in Thoth, these factors make for a small, efficient kernel.

The V message primitives appear ill-suited in several ways for a network environment, at least on first observation. The short, fixed-length messages appear to make inefficient use of large packet sizes typically available on local networks. The synchronous nature of the message sending would seem to interfere with the true parallelism possible between separate workstations. And the economies of message buffering afforded by these restrictions in a single machine implementation are less evident in a distributed environment. Finally, the separate data transfer operations ***MoveTo*** and ***MoveFrom*** appear only to increase the number of remote data transfer operations that must be implemented in the distributed case.

However, our experience has been that the V message primitives are easily and efficiently implemented over a local network. Moreover, we have found that the semantics of the primitives facilitated an efficient distributed implementation. The only major departure from Thoth was the explicit specification of segments in messages and the addition of the primitives ***ReceiveWithSegment*** and ***ReplyWithSegment***. This extension was done for efficient page-level file access although it has proven useful under more general circumstances, e.g. in passing character string names to name servers.

3. Implementation Issues

A foremost concern in the implementation of the kernel has been efficiency. Before describing some of the implementation details of the individual primitives, we list several aspects of the implementation that are central to the efficient operation of the kernel.

1. Remote operations are implemented directly in the kernel instead of through a process-level network server. When the kernel recognizes a request directed to a remote process, it immediately writes an appropriate packet on the network. The alternative approach whereby the kernel relays a remote request to a network server who then proceeds to write the packet out on the network incurs a heavy penalty in extra copying and process switching. (We measured a factor of four increase in the remote message exchange time.)
2. Interkernel packets use the "raw" Ethernet data link level. The overhead of layered protocol implementation has been described many times [10]. An alternative implementation using internet (IP) headers showed a 20 percent increase in the basic message exchange time, even without computing the IP header checksum and with only the simplest routing in the kernel. While we recognize the tradeoff between internet functionality and local net performance, we have chosen not to burden the dominant (local net) operation with any more overhead than is strictly necessary.
3. The synchronous request-response nature of a reply associated with each message is exploited to build reliable message transmission directly on an unreliable datagram service, i.e. without using an extra layer (and extra packets) to implement reliable transport. The reply message serves as an acknowledgement as well as carrying the reply message data.
4. The mapping from process id to process location is aided by encoding a host specification in the process identifier. The kernel can thus determine quickly whether a process is either local or remote, and in the latter case on which machine it resides.
5. There are no per-packet acknowledgements for large data transfers (as in ***MoveTo*** and in ***MoveFrom***). There is only a single acknowledgement when the transfer is complete.
6. File page-level transfers require the minimal number of packets (i.e. two) because of the ability to append short segments to messages using ***ReceiveWithSegment*** and ***ReplyWithSegment***.

The following sections look at particular aspects of the implementation in greater detail.

3.1. Process Naming

V uses a global (flat) naming space for specifying processes, in contrast to the local port naming used in DEMOS [1] and Accent [12]. Process identifiers are unique within the context of a local network. On the SUN workstation, it is natural for the V kernel to use 32-bit process identifiers. The high-order 16 bits of the process identifier serve as a *logical host identifier* subfield while the low-order 16 bits are used as a locally unique identifier.

In the current 3 Mb Ethernet implementation, the top 8 bits of the logical host identifier are the physical network address of the workstation, making the process identifier to network address mapping trivial. In the 10 Mb implementation, a table maps logical hosts to network addresses. When there is no table entry for the specified logical host, the message is broadcast. New "logical host-to-network address" correspondences can be discovered from messages received. However, each node must at least know or discover its own logical host identifier during kernel initialization.

The use of an explicit host field in the process identifier allows distributed generation of unique process identifiers between machines and allows an efficient mapping from process id to network address. In particular, it is very efficient to determine whether a process is local or remote. This "locality" test on process identifiers serves as the primary invocation mechanism from the local kernel software into the network IPC portion. In general, most V kernel operations differ from their Thoth implementation by a call to a "non-local" handler when one of the process identifier parameters fails to validate as a local

process. With the exception of *GetPid*, kernel operations with no process identifier parameters are implicitly local to the workstation.

GetPid uses network broadcast to determine the mapping of a logical process identifier to real process identifier if the mapping is not known to the local kernel. Any kernel knowing the mapping can respond to the broadcast request. The addition of local and remote scopes was required to discriminate between server processes that serve only a single workstation and those that serve the network.

3.2. Remote Message Implementation

When a process identifier is specified to *Send* with a logical host identifier different from that of the local machine, the local pid validation test fails and *Send* calls *NonLocalSend* which handles transmission of the message over the network.

The *NonLocalSend* routine writes a interkernel packet on the network addressed to the host machine of this process or else broadcasts the packet if the host machine is not known. When the host containing the recipient process receives the packet, it creates an *alien* process descriptor to represent the remote sending process using a standard kernel process descriptor[1] and saves the message in the message buffer field of the alien process descriptor. When the receiving process replies to the message, the reply is transmitted back to the sender as well as being saved for a period of time in the alien descriptor. If the sender does not receive a reply within the timeout period T, the original message is retransmitted by the sender's kernel. The receiving kernel filters out retransmissions of received messages by comparing the message sequence number and source process with those represented by the aliens. The kernel responds to a retransmitted message by discarding the message and either retransmitting the reply message or else sending back a "reply-pending" packet to the sending kernel if the reply has not yet been generated. It also sends back a reply-pending packet if it is forced to discard a new message because no (alien) process descriptors are available. The sending kernel concludes the receiving process does not exist (and thus the *Send* has failed) if it receives a negative acknowledgement packet or it retransmits N times without receiving either a reply message or a reply-pending packet.

This description supports the claim made above that reliable message transmission is built immediately on top of an unreliable datagram protocol with the minimal number of network packets in the normal case.

3.3. Remote Data Transfer

MoveTo and *MoveFrom* provide a means of transferring a large amount of data between remote processes with a minimal time increase over the time for transferring the same amount of data in raw network datagrams. *MoveTo* transmits the data to be moved in a sequence of maximally-sized packets to the destination workstation and awaits a single acknowledgement packet when all the data has been received. Given the observed low error rates of local networks, full retransmission on error introduces only a slight performance degradation. We have, however, implemented retransmission from the last correctly received data packet in order to avoid the pitfall of repeated identical failures arising when back-to-back packets are consistently being dropped by the receiver. The implementation of *MoveFrom* is similar except a *MoveFrom* request is sent out and acknowledged by the requested data packets, essentially the reverse of *MoveTo*.

As in the local case, major economies arise with *MoveTo* and *MoveFrom* because, by their definitions, there is no need for queuing or buffering of the data in the kernel. The V kernel moves the data directly from the source address space into the network interface, and directly from the network interface to the destination address space[2].

3.4. Remote Segment Access

The message and data transfer primitives provide efficient communication of small amounts and large amounts of data, less than 32 bytes or several tens of network packets. However, page-level file access requests involve an intermediate amount of data that is not efficiently handled by the Thoth primitives when implemented over a local network.

V file access is implemented using an I/O protocol developed for Verex [4]. To read a page or block of a file, a client sends a message to the file server process specifying the file, block number, byte count and the address of the buffer into which the data is to be returned. The server reads the data off disk, transfers it into the remote buffer using *MoveTo*, and then replies, confirming the amount of data read. In the common case of the data fitting into a single network packet, this requires 4 packet transmissions (assuming no retransmissions): one for the *Send*, 2 for the *MoveTo* and one for the *Reply*. This is double the number of packets required by a specialized page-level file access protocol as used, for instance, in LOCUS [11] or WFS [14].

To remedy this problem, we made some modifications to the original Thoth kernel interface. First, we added the primitives *ReceiveWithSegment* and *ReplyWithSegment*. Second, we required explicit specification of segments in messages (as described in Section 2.1). Using this explicit specification, *Send* was then modified to transmit, as part of the network packet containing the message, the first part of a segment to which read access has been granted, if any. Using the *ReceiveWithSegment* operation, the recipient process is able to receive the request message and the first portion of the segment in a single operation. By setting the size of the initial portion of the segment sent to be

[1] Use of standard kernel process descriptors for aliens reduces the amount of specialized code for handling remote messages. However, alien processes do not execute and can reasonably be thought of as message buffers.

[2] This is possible with a programmed I/O interface, as used by the SUN 3 Mb Ethernet interface as well as the 3COM 10 Mb Ethernet interface. A conventional DMA interface may require a packet assembly buffer for transmission and reception.

at least as large as a file block, a file write operation is reduced to a single two packet exchange. In this fashion, read access to small segments (such as single block disk write operations) is handled efficiently. The *ReplyWithSegment* operation eliminates the extra packets in the case of a file read by combining the reply message packet with the data to be returned.

Advantages of this approach include:

1. The advantages of the Thoth IPC model with the network performance of a WFS-style file access protocol.
2. Compatibility with efficient local operation. For instance, segments may be accessed directly if in the same address space or if the recipient process operates a DMA device.
3. Use of *ReceiveWithSegment* and *ReplyWithSegment* is optional and transparent to processes simply using *Send*.

An expected objection to our solution is the apparent asymmetry and awkwardness of the message primitives. We feel the asymmetry may be appropriate given the prevalent asymmetry of communication and sophistication between client and server processes. Also, it is not unexpected that there be some conflict between efficiency and elegance. Given that applications commonly access system services through stub routines that provide a procedural interface to the message primitives, it is not inappropriate to make some compromises at the message level for efficiency.

We now turn to the discussion of the performance evaluation of the kernel. We first define the term *network penalty* as a reasonable lower bound on the cost of network communication. Subsequently we discuss the efficiency of the kernel, both in terms of message passing and file access.

4. Network Penalty

Our measurements of the V kernel are primarily concerned with two comparisons:

1. The cost of remote operations versus the cost of the corresponding local operations.
2. The cost of file access using V kernel remote operations versus the cost for other means of network file access.

An important factor in both comparisons is the cost imposed by network communication. In the first comparison, the cost of a remote operation should ideally be the cost of the local operation plus the cost of moving data across the network (data that is in shared kernel memory in the local case). For instance, a local message *Send* passes pointers to shared memory buffers and descriptors in the kernel while a remote message *Send* must move the same data across the network. In the second comparison, the basic cost of moving file data across the network is a lower bound on the cost for any network file access method.

To quantify the cost of network communication, we define a measure we call the *network penalty*. The network penalty is defined to be the time to transfer n bytes from one workstation to another in a network datagram on an idle network and assuming no errors. The network penalty is a function of the processor, the network, the network interface and the number of bytes transferred. It is the minimal time penalty for interposing the network between two software modules that could otherwise transmit the data by passing pointers. The network penalty is obtained by measuring the time to transmit n bytes from the main memory of one workstation to the main memory of another and vice versa and dividing the total time for the experiment by 2. The experiment is repeated a large number of times for statistical accuracy. The transfers are implemented at the data link layer and at the interrupt level so that no protocol or process switching overhead appears in the results. The assumptions of error-free transmission and low network utilization are good approximations of most local network environments.

Network penalty provides a more realistic minimum achievable time for data transfer than that suggested by the physical network speed because it includes the processor and network interface times. For instance, a 10 Mb Ethernet can move 1000 bits from one workstation to another in 100 microseconds. However, the time to assemble the packet in the interface at the sending end and the time to transfer the packet out of the interface at the receiving end are comparable to the time for transmission. Thus, the time for the transfer from the point of view of the communicating software modules is at least two or three times as long as that implied by the physical transfer rate.

Measurements of network penalty were made using the experimental 3 Mb Ethernet. In all measurements, the network was essentially idle due to the unsociable times at which measurements were made. Table 4-1 lists our measurements of the 3 Mb network penalty for the SUN workstation using the 8 and 10 MHz processors with times given in milliseconds. The network time column gives the time for the data to be transmitted based on the physical bit rate of the medium, namely 2.94 megabits per second.

Network Penalty

Bytes[3]	Network Time	Network Penalty	
		8 MHz	10 MHz
64	.174	0.80	0.65
128	.348	1.20	0.96
256	.696	2.00	1.62
512	1.392	3.65	3.00
1024	2.784	6.95	5.83

Table 4-1: 3 Mb Ethernet SUN Network Penalty (times in msec.)

The network penalty for the 8 MHz processor is roughly given by $P(n) = .0064 * n + .390$ milliseconds where n is the number of bytes transferred. For the 10 MHz processor, it is $.0054 * n + .251$ milliseconds.

[3] We only consider packet sizes that fit in a single network datagram.

The difference between the network time, computed at the network data rate, and the measured network penalty time is accounted for primarily by the processor time to generate and transmit the packet and then receive the packet at the other end. For instance, with a 1024 byte packet and an 8 MHz processor, the copy time from memory to the Ethernet interface and vice versa is roughly 1.90 milliseconds in each direction. Thus, of the total 6.95 milliseconds, 3.80 is copy time, 2.78 is network transmission time and .3 is (presumably) network and interface latency. If we consider a 10 Mb Ethernet with similar interfaces, the transmission time is less than 1 millisecond while the copy times remain the same, making the processor time 75 percent of the cost in the network penalty. The importance of the processor speed is also illustrated by the difference in network penalty for the two processors measured in Table 4-1.

With our interfaces, the processor is required to copy the packet into the interface for transmission and out of the interface on reception (with the interface providing considerable on-board buffering). From the copy times given above, one might argue that a DMA interface would significantly improve performance. However, we would predict that a DMA interface would not result in better kernel performance for two reasons. First, the kernel interprets a newly arrived packet as it copies the packet from the network interface, allowing it to place much of the packet data immediately in its final location. With a DMA interface, this copy would be required after the packet had been DMA'ed into main memory. Similarly, on transmission the kernel dynamically constructs a network packet from disparate locations as it copies the data into the network interface. Most DMA interfaces require the packet to be assembled in one contiguous area of memory, implying the need for a comparable copy operation. Finally, there is not currently available (to our knowledge) a network DMA interface for the Ethernet that moves data faster than a 10 MHz Motorola 68000 processor as used in the SUN workstation. In general, the main benefit of a smart network interface appears to be in offloading the main processor rather than speeding up operations that make use of network communication.

5. Kernel Performance

Our first set of kernel measurements focuses on the speed of local and network interprocess communication. The kernel IPC performance is presented in terms of the times for message exchanges and the data transfer operations. We first describe how these measurements were made.

5.1. Measurement Methods

Measurements of individual kernel operations were performed by executing the operation N times (typically 1000 times), recording the total time required, subtracting loop overhead and other artifact, and then dividing the total time by N. Measurement of total time relied on the software-maintained V kernel time which is accurate plus or minus 10 milliseconds.

Measurement of processor utilization was done using a low-priority "busywork" process on each workstation that repeatedly updates a counter in an infinite loop. All other processor utilization reduces the processor allocation to this process. Thus, the processor time used per operation on a workstation is the elapsed time minus the processor time allocated to the "busywork" process divided by N, the number of operations executed.

Using 1000 trials per operation and time accurate plus or minus 10 milliseconds, our measurements should be accurate to about .02 milliseconds except for the effect of variation in network load.

5.2. Kernel Measurements

Table 5-1 gives the results of our measurements of message exchanges and data transfer with the kernel running on workstations using an 8 MHz processor and connected by a 3 Mb Ethernet. Note that *GetTime* is a trivial kernel operation, included to give the basic minimal overhead of a kernel operation. The columns labeled Local and Remote give the elapsed times for these operations executed locally and remotely. The Difference column lists the time difference between the local and remote operations. The Penalty column gives the network penalty for the amount of data transmitted as part of the remote operation. The Client and Server columns list the processor time used for the operations on the two machines involved in the remote execution of the operation. Table 5-2 gives the same measurements using a 10 MHz processor. The times for both processors are given to indicate the effect the processor speed has on local and remote operation performance. As expected, the times for local operations, being dependent only on the processor speed, are 25 percent faster on the 25 percent faster processor. However, the almost 15 percent improvement for remote operations indicates the processor is the most significant performance factor in our configuration and is not rendered insignificant by the network delay (at least on a lightly loaded network).

A significant level of concurrent execution takes place between workstations even though the message-passing is fully synchronized. For instance, transmitting the packet, blocking the sender and switching to another process on the sending workstation proceeds in parallel with the reception of the packet and the readying of the receiving process on the receiving workstation. Concurrent execution is indicated by the fact that the total of the server and client processor times is greater than the elapsed time for a remote message exchange. (See the Client and Server columns in the above tables.)

5.3. Interpreting the Measurements

Some care is required in interpreting the implications of these measurements for distributed applications. Superficially, the fact that the remote *Send-Receive-Reply* sequence takes more than 3 times as long as for the local case suggests that distributed applications should be designed to minimize inter-machine communication. In general, one might consider it impractical to

Kernel Performance

Kernel Operation	Elapsed Time			Network Penalty	Processor Time	
	Local	Remote	Difference		Client	Server
GetTime	0.07	-	-	-	0.07	-
Send-Receive-Reply	1.00	3.18	2.18	1.60	1.79	2.30
MoveFrom: 1024 bytes	1.26	9.03	7.77	8.15	3.76	5.69
MoveTo: 1024 bytes	1.26	9.05	7.79	8.15	3.59	5.87

Table 5-1: Kernel Performance: 3 Mb Ethernet and 8 MHz Processor (times in milliseconds)

Kernel Operation	Elapsed Time			Network Penalty	Processor Time	
	Local	Remote	Difference		Client	Server
GetTime	0.06	-	-	-	0.06	-
Send-Receive-Reply	0.77	2.54	1.77	1.30	1.44	1.79
MoveFrom: 1024 bytes	0.95	8.00	7.05	6.77	3.32	4.78
MoveTo: 1024 bytes	0.95	8.00	7.05	6.77	3.17	4.95

Table 5-2: Kernel Performance: 3 Mb Ethernet and 10 MHz Processor (times in milliseconds)

view interprocess communication as transparent across machines when the speed ratio is so large. However, an alternative interpretation is to recognize that the remote operation adds a delay of less than 2 milliseconds, and that in many cases this time is insignificant relative to the time necessary to process a request in the server. Furthermore, the sending or client workstation processor is busy with the remote ***Send*** for only 1.44 milliseconds out of the total 2.54 millisecond time (using the 10 MHz processor). Thus, one can offload the processor on one machine by, for instance, moving a server process to another machine if its request processing generally requires more than 0.67 milliseconds of processor time, i.e. the difference between the local ***Send-Receive-Reply*** time and the local processor time for the remote operation.

5.4. Multi-Process Traffic

The discussion so far has focused on a single pair of processes communicating over the network. In reality, processes on several workstations would be using the network concurrently to communicate with other processes. Some investigation is required to determine how much message traffic the network can support and also the degradation in response as a result of other network traffic.

A pair of workstations communicating via ***Send-Receive-Reply*** at maximum speed generate a load on the network of about 400,000 bits per second, about 13 percent of a 3 Mb Ethernet and 4 percent of a 10 Mb Ethernet. Measurements on the 10 Mb Ethernet indicate that for the packet size in question no significant network delays are to be expected for loads upto 25 percent. Thus, one would expect minimal degradation with say two separate pairs of workstations communicating on the same network in this fashion. Unfortunately, our measurements of this scenario turned up a hardware bug in our 3 Mb Ethernet interface, a bug which causes many collisions to go undetected and show up as corrupted packets. Thus, the response time for the 8 MHz processor workstation in this case is 3.4 milliseconds. The increase in time from 3.18 milliseconds is accounted for almost entirely from the timeouts and retransmissions arising (roughly one per 2000 packets) from the hardware bug. With corrected network interfaces, we estimate that the network can support any reasonable level of message communication without significant performance degradation.

A more critical resource is processor time. This is especially true for machines such as servers that tend to be the focus of a significant amount of message traffic. For instance, just based on server processor time, a workstation is limited to at most about 558 message exchanges per second, independent of the number of clients. The number is substantially lower for file access operations, particularly when a realistic figure for file server processing is included. File access measurements are examined in the next section.

6. File Access Using the V Kernel

Although it is attractive to consider the kernel as simply providing message communication, a predominant use of the message communication is to provide file access, especially in our environment of diskless personal workstations. File access takes place in several different forms, including: random file page access, sequential file reading and program loading. In this section, we assume the file server is essentially dedicated to

serving the client process we are measuring and otherwise idle. A later section considers multi-client load on the file server.

We first describe the performance of random page-level file access.

6.1. Page-level File Access

Table 6-1 list the times for reading or writing a 512 byte block between two processes both local and remote using the 10 MHz processor. The times do not include time to fetch the data from disk but do indicate expected performance when data is buffered in memory. A page read involves the sequence of kernel operations: ***Send--Receive--ReplyWithSegment.*** A page write is ***Send--ReceiveWithSegment--Reply.***

difference between the client processor time for remote page access and for local page access, namely 1.3 milliseconds. A processor cost of more than 1.3 milliseconds per request can be expected from the estimation made earlier using LOCUS figures.

These measurements indicate the performance when file reading and writing use explicit segment specification in the message and ***ReceiveWithSegment*** and ***ReplyWithSegment.*** However, a file write can also be performed in a more basic Thoth-like way using the ***Send-Receive-MoveFrom-Reply*** sequence. For a 512 byte write, this costs 8.1 milliseconds; file reading is similar using ***MoveTo.*** Thus, the segment mechanism saves 3.5 milliseconds on every page read and write operation, justifying this extension to the message primitives.

Random Page-Level Access

	Elapsed Time			Network Penalty	Processor Time	
Operation	Local	Remote	Difference		Client	Server
page read	1.31	5.56	4.25	3.89	2.50	3.28
page write	1.31	5.60	4.29	3.89	2.58	3.32

Table 6-1: Page-Level File Access: 512 byte pages (times in milliseconds)

The columns are to be interpreted according to the explanation given for similarly labeled columns of Tables 5-1 and 5-2. Note that the time to read or write a page using these primitives is approximately 1.5 milliseconds more than the network penalty for these operations.

There are several considerations that compensate for the cost of remote operations being higher than local operations. (Some are special cases of those described for simple message exchanges.) First, the extra 4.2 millisecond time for remote operations is relatively small compared to the time cost of the file system operation itself. In particular, disk access time can be estimated at 20 milliseconds (assuming minimal seeking) and file system processor time at 2.5 milliseconds.[4] This gives a local file read time of 23.8 milliseconds and a remote time of 28.1 milliseconds, making the cost of the remote operation 18 percent more than the local operation.

This comparison assumes that a local file system workstation is the same speed as a dedicated file server. In reality, a shared file server is often faster because of the faster disks and more memory for disk caching that come with economy of scale. If the average disk access time for a file server is 4.3 milliseconds less than the average local disk access time (or better), there is no time penalty (and possibly some advantage) for remote file operations.

Second, remote file access offloads the workstation processor if the file system processing overhead per request is greater the

[4]This is based on measurements of LOCUS [11] that give 6.2 and 4.3 milliseconds as processor time costs for 512-byte file read and write operations respectively on a PDP-11/45, which is roughly half the speed of the 10 MHz Motorola 68000 processor used in the SUN workstation.

6.2. Sequential File Access

Sequential file access is the predominant pattern for file activity in most systems. Efficient file systems exploit this behavior to reduce the effect of disk latency by prefetching file pages (read-ahead) and asynchronously storing modified pages (write-behind). File access and file transfer protocols typically implement streaming to reduce the effect of network latency on sequential file access.

Using the V kernel message communication between a workstation and a file server, the file server can implement read-ahead and write-behind to reduce the effect of disk latency. However, there is no streaming in the network IPC to deal with network latency.

Two factors suggest that streaming can be done without in a local network environment. First, local networks have a low latency as a consequence of their high speed and low delay. Second, although V kernel IPC is logically synchronous, significant concurrency arises in the network implementation, further reducing the effects of network latencies. The presence of streaming adds a significant cost to the protocol in terms of buffer space, copying time and complexity of code. Moreover, buffering effectively puts a disk cache on the workstation, thus raising problems of cache consistency between the different workstations and the file server.

To get a realistic measure of sequential file access performance, we modified the test program used for measuring page read times by the addition of a delay in the server process corresponding to the disk latency. Because we assume the file server is doing read-ahead operations, the delay is interposed between the reply to one

request and the receipt of the next request. We contend that this program closely approximates the performance of a workstation sequentially reading a file as rapidly as possible from an otherwise idle file server. The results are shown in Table 6-2

Sequential Page-Level Access

Disk Latency	Elapsed Time per Page Read
10	12.02
15	17.13
20	22.22

Table 6-2: Seq. File Access: 512 byte pages (times in msec.)

These results indicate that, for reasonable values of disk latency, the elapsed time for sequential file access is within 10 to 15 percent from the minimum achievable (the disk latency). It follows that a streaming protocol cannot improve on the performance measured for V kernel file access by more than 15 percent.

Moreover, consider the two cases for the application, namely: Reading faster than the disk latency and reading slower than the disk latency. Suppose an application is reading file pages over the network using a streaming protocol. If it is reading faster than the disk can deliver, it will operate much like the V kernel model in attempting to read a page not yet available, possibly requesting this page from the file server, blocking waiting for the page, having the page returned into the local page buffers, copying the page into its application buffer and then continuing. Thus, performance of a streaming file access implementation can be expected to be similar to our results. For instance, comparing our results with the LOCUS figures for remote sequential file access with a disk latency of 15 milliseconds, the average elapsed time per page is essentially equal to the LOCUS figure of 17.18 milliseconds.

If an application is reading pages sequentially slower than the disk latency time, with say 20 milliseconds between every read request, the page should be available locally on each read with a streaming protocol. In this case, the elapsed time for the read should be 1.3 milliseconds compared to 5.6 milliseconds remotely. However, because read operations occur at most every 20 milliseconds, the speedup by replacing non-streamed file access with streamed file access is limited to 20 percent or less.

Moreover, a streaming protocol would introduce extra processing overhead for copying and buffering readahead pages in this circumstance. Assuming the application was reading slowly because it was compute-bound between read operations, the streaming protocol processing overhead would further slow down the application From this analysis, it is clear that streaming has limited potential for speedup over non-streamed file access when pages are accessed from disk with the latencies we have discussed.

In most systems, sequential file access is used extensively for program loading. However, program loading can be performed more efficiently with the V kernel using ***MoveTo***. It is therefore not reliant on the performance figures of this section and is discussed below.

6.3. Program Loading

Program loading differs as a file access activity from page-level access in that the entire file containing the program (or most of it) is to be transferred as quickly as possible into a waiting program execution space. For instance, a simple command interpreter we have written to run with the V kernel loads programs in two read operations: the first read accesses the program header information; the second read copies the program code and data into the newly created program space. The time for the first read is just the single block remote read time given earlier. The second read, generally consisting of several tens of disk pages, uses ***MoveTo*** to transfer the data. Because ***MoveTo*** requires that the data be stored contiguously in memory, it is often convenient to implement a large read as multiple ***MoveTo*** operations. For instance, our current VAX file server breaks large read and write operations into ***MoveTo*** and ***MoveFrom*** operations of at most 4 kilobytes at a time. Table 6-3 gives the time for a 64 kilobyte Read. (The elapsed time for file writing is basically the same as for reading and has been omitted for the sake of brevity. Note also that network penalty is not defined for multi-packet transfers.) The transfer unit is the amount of data transferred per ***MoveTo*** operation in satisfying the read request.

The times given for program loading using a 16 or 64 kilobyte transfer unit corresponds to a data rate of about 192 kilobytes per second, which is within 12 percent of the data rate we can achieve on a SUN workstation by simply writing packets to the network interface as rapidly as possible. Moreover, if the file server retained copies of frequently used programs in memory, much as

Program Loading

	Elapsed Time			Network Penalty	Processor Time	
Transfer unit	Local	Remote	Difference		Client	Server
1 Kb	71.7	518.3	446.5	434.5	207.1	297.9
4 Kb	62.5	368.4	305.8	*	176.1	225.2
16 Kb	60.2	344.6	284.3	*	170.0	216.9
64 Kb	59.7	335.4	275.1	*	168.1	212.7

Table 6-3: 64 kilobyte Read (times in milliseconds)

many current timesharing systems do, such program loading could achieve the same performance given in the table, independent of disk speed. Thus, we argue that *MoveTo* and *MoveFrom* with large transfer units provide an efficient program loading mechanism that is as fast as can be achieved with the given hardware.

7. File Server Issues

File server performance is a critical issue for diskless workstations. Unfortunately, we do not yet have experience with a V kernel-based file server. Thus, this section describes what we believe are the key issues and estimates performance without providing conclusive data. In general, we view the processor as the key resource to consider in file server performance because, as argued earlier, the network bandwidth is plentiful and disk scheduling and buffering issues are identical to those encountered in conventional multi-user systems.

The number of workstations a file server can support can be estimated from processor requirements. If we estimate page read or write processing overhead as roughly 3.5 milliseconds for file system processing (from LOCUS) plus 3.3 milliseconds for kernel operation (from Table 6-1]), a page request costs about 7 milliseconds of processor time. Program loading appears to cost about 300 milliseconds for an average 64 kilobyte program. Estimating that 90 percent of the file requests are page requests, the average request costs 36 milliseconds. Thus, a file server based on the SUN workstation processor could support about 28 file requests a second. From this we estimate that one file server can serve about 10 workstations satisfactorily, but 30 or more active workstations would lead to excessive delays. However, a diskless workstation system can easily be extended to handle more workstations by adding more file server machines since the network would not seem to be a bottleneck for less than 100 workstations.

For some programs, it is advantageous in terms of file server processor requirements to execute the program on the file server, rather than to load the program into a workstation and subsequently field remote page requests from it. Large programs, executing for a short time and doing a lot of file access while executing are in this class, especially if they require only limited interaction with the user.

On this basis, a file server should have a general program execution facility and the ability to selectively execute certain programs. The need for this execution facility is a further argument for using a general interprocess communication mechanism in place of a specialized page-level file access protocol. With the V kernel, all inter-program communication and interaction takes place through the IPC facility, including: file access, argument passing, debugger control and termination notification. Thus, execution of a program in the file server rather than the client's workstation does not change the program's execution environment nor the client's interaction with the program, i.e. it is transparent except for performance.

8. Measurements with the 10 Mb Ethernet

Our limited access to a 10 Mb Ethernet has precluded basing our measurements on this standard local network. However, some preliminary figures using the 10 Mb Ethernet indicate the effect of using a faster network and slightly faster network interfaces. First, the remote message exchange time is 2.71 milliseconds using an 8 MHz processor, roughly the time for the 10 MHz processor on the 3 Mb network and .5 milliseconds better than the 8 MHz processor on the 3 Mb network. Second, the page read time is 5.72 milliseconds. Finally, the program loading time is much improved, achieving 255 milliseconds for a 64 kilobyte load using 16 Kb transfer units. We have not identified to what degree the improvement is due to the faster network speed versus the differences in the network interface.

9. Related Work

There are a number of previous and concurrent efforts in providing communication mechanisms for distributed systems. For brevity, we compare our work with only a representative sample that characterizes the search for, and evaluation of, suitable models and implementations.

Spector's remote reference study [13] considered the feasibility of implementing remote load and store operations over a local network. Nelson's work on remote procedure calls [10] investigates network communication for procedure-based systems analogous to what the V kernel provides for message-based systems. Rashid and Robertson's Accent kernel [12] implements a message system with a number of features such as non-blocking message sending that are not provided by the V kernel. Finally, LOCUS [11] integrates network communication into a UNIX-like system in the form of transparent remote file access.

Our work has followed the pattern of Spector's and Nelson's work in using a *request-response* form of communication (in place of streaming) and stripping away protocol layers for adequate performance. However, we share Spector's concern about the utility of an unprotected globally shared memory in a distributed system, which is essentially the functionality provided by his primitives. The V kernel provides a strong degree of separation between processes and supports protected provision of services in a multi-user, multi-workstation environment by limiting interprocess communication to the kernel IPC primitives.

Our approach differs from Nelson's primarily in our use of a separate interconnection mechanism from procedure calls, namely messages, and some of the ensuing differences in semantics. Fundamentally, the V kernel provides a base on which to build a remote procedure call mechanism by the addition of suitable compiler and run-time support. Under more detailed examination, many slight differences appear that reflect long-established differences in the construction of procedure-based

versus message-based systems, although it is not clear these differences result in any significant difference in overall performance.

The V kernel performance is roughly comparable to that of the software implementations developed by Spector and Nelson, allowing for the non-trivial differences in operation semantics and host processors. We would hypothesize that V kernel performance could be improved by a factor of 30 using microcode, similar to the improvement observed by Spector and Nelson for their primitives. Unfortunately, neither Spector nor Nelson provides results that afford a comparison with our file access results. In general, their work has concentrated on the speed of the basic mechanism and has not been extended to measure performance in a particular application setting.

In comparison to Accent, the V kernel provides a primitive form of message communication, and benefits accordingly in terms of speed, small code size and ability to run well on an inexpensive machine[5] without disk or microcode support. For instance, Accent messages require an underlying transport protocol for reliable delivery because there is no client-level reply message associated with every *Send* as in the V kernel. We do not at this time have performance figures for Accent.

LOCUS does not attempt to provide applications with general network interprocess communication but exploits carefully honed problem-oriented protocols for efficient remote file access. It is difficult to compare the two systems from measurements available given the differences in network speeds, processor speeds and measurement techniques. However, from the specific comparisons with LOCUS presented earlier, we would expect overall file access performance for the V kernel to be comparable to LOCUS running on the same machines and network.

However, the memory requirements for the V kernel are about half that of LOCUS compiled for the PDP-11 and probably more like one fourth when LOCUS is compiled for a 32-bit processor like the 68000. Thus, for graphics workstations or process control applications, for instance, the V kernel would be more attractive because of its smaller size, real-time orientation and its provision of general interprocess communication. However, the V kernel does not provide all the functionality of the LOCUS kernel which includes that of the UNIX kernel and more. When required with V, these additional facilities must be provided by server processes executing either on client workstations or network server machines.

10. Conclusions

We conclude that it is feasible to build a distributed system using diskless workstations connected by a high-speed local network to one or more file servers using the V kernel IPC. In particular, the performance study shows that V kernel IPC provides satisfactory performance despite its generality. Because the performance is so close to the lower bound given by the network penalty, there is relatively little room for improvement on the V IPC for the given hardware regardless of protocol and implementation used.

The efficiency of file access using the V IPC suggests that it can not only replace page-level file access protocols but also file transfer and remote terminal protocols, thereby reducing the number of protocols needed. We claim that V kernel IPC is adequate as a transport level for all our local network communication providing each machine runs the V kernel or at least handles the interkernel protocol. We do, however, see a place for these specific protocols in internetworking situations.

In addition to quantifying the elapsed time for various operations, our study points out the importance of considering processor requirements in the design of distributed systems. More experience and measurement of file server load and workstation file access behavior is required to decide whether file server processing is a significant problem in using diskless workstations.

The V kernel has been in use with the diskless SUN workstations, providing local and remote interprocess communication, since September 1982. It is currently 38 kilobytes including code, data and stack. The major use of the network interprocess communication is for accessing remote files. Our file servers are currently 6 VAX/UNIX systems running a kernel simulator and file server program which provides access to UNIX system services over the Ethernet using interkernel packets. A simple command interpreter program allows programs to be loaded and run on the workstations using these UNIX servers. Our experience with this software to date supports the conclusions of the performance study that we can indeed build our next generation of computing facilities [8] using diskless workstations and the V kernel.

[5] A diskless SUN workstation is much less than the cost of a PERQ.

Acknowledgements

We are indebted to all the members of the V research group at Stanford, which at this point includes two faculty members and roughly ten graduate students. In particular, we wish to thank Keith Lantz for his patient comments on a seemingly endless sequence of drafts and Tim Mann for his many contributions to the design and the implementation of the kernel. We would also like to thank the referees whose comments and suggestions helped to enhance the clarity of the paper.

References

1. F. Baskett, J.H. Howard and J.T. Montague. Task Communication in DEMOS. Proceedings of the 6th Symposium on Operating System Principles, ACM, November, 1977, pp. 23-31. Published as *Operating Systems Review* 11(5).

2. A. Bechtolsheim, F. Baskett, V. Pratt. The SUN Workstation Architecture. Tech. Rept. 229, Computer Systems Laboratory, Stanford University, March, 1982.

3. D.R. Cheriton, M.A. Malcolm, L.S. Melen and G.R. Sager. "Thoth, a Portable Real-time Operating System." *Comm. ACM 22*, 2 (February 1979), 105-115.

4. D.R. Cheriton. Distributed I/O using an Object-based Protocol. Tech. Rept. 81-1, Computer Science, University of British Columbia, 1981.

5. D.R. Cheriton. *The Thoth System: Multi-process Structuring and Portability.* American Elsevier, 1982.

6. D.R. Cheriton, T.P. Mann and W. Zwaenepoel. V-System: Kernel Manual. Computer Systems Laboratory, Stanford University

7. Digital Equipment Corporation, Intel Corporation and Xerox Corporation. The Ethernet: A Local Area Network - Data Link Layer and Physical Layer Specifications, Version 1.0.

8. K.A. Lantz, D.R. Cheriton and W.I. Nowicki. Third Generation Graphics for Distributed Systems. Tech. Rept. STAN-CS-82-958, Department of Computer Science, Stanford University, February, 1983. To appear in *ACM Transactions on Graphics*

9. R.M. Metcalfe and D.R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Comm. ACM 19*, 7 (July 1976), 395-404. Also CSL-75-7, Xerox Palo Alto Research Center, reprinted in CSL-80-2.

10. B.J. Nelson. *Remote Procedure Call.* Ph.D. Th., Carnegie-Mellon U., 1981. published as CMU technical report CMU-CS-81-119

11. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G.Rudisin, G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. Proceedings of the 8th Symposium on Operating Systems Principles, ACM, December, 1981, pp. 169-177.

12. R. Rashid and G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. Proceedings of the 8th Symposium on Operating Systems Principles, ACM, December, 1981, pp. 64-75.

13. A. Spector. "Performing Remote Operations Efficiently on a Local Computer Network." *Comm. ACM 25*, 4 (April 1982), 246-260.

14. D. Swinehart, G. McDaniel and D. Boggs. WFS: A Simple Shared File System for a Distributed Environment. Proceedings of the 7th Symposium on Operating Systems Principles, ACM, December, 1979, pp. 9-17.