Using Hints in DUNE Remote Procedure Calls

Marc F. Pucci and J. L. Alberi Bellcore

ABSTRACT: Remote Procedure Calls (RPC) are an established mechanism for coordinating activity between multiple processors in a distributed system. Their efficient use in Interprocess Communication (IPC) is difficult because underlying network protocols can have a great effect on system performance, especially when the IPC is within the same processor. This problem is compounded if the distributed system must work simultaneously over several forms of interconnect. Designing an efficient solution for one medium may make accommodating another awkward or impractical, while designing a general solution for all may not take full advantage of any specific medium.

This paper describes an enhanced remote procedure call model of interprocess communication that is generally efficient for many types of network, even nontraditional ones. It is based on optional *hints* supplied by the client of an operation, which enable medium-specific optimizations in a uniformly structured interface. The model supports dynamic rebinding of communications protocols, which is necessary for the efficient treatment of migratable system objects.

1. Introduction

Distributed operating systems running on native hardware often rely on special purpose protocols for interprocess communication [Accetta et al. 1986; Cheriton 1984; Popek et al. 1981].[1] Special protocols provide efficient operation by reducing the layering inherent in general purpose protocols, but are difficult to adapt to new network technology [Hutchinson et al. 1989]. Systems that require several special purpose protocols to support different communications media are difficult to implement and maintain because of the complexity of the code. General-purpose layered protocols [DOD 1980; ISO 1982]. provide easy adaptation to new technology and support for multiple networks, but are slow, especially when each layer is mapped into a process [Clark 1985].

The DUNE distributed operating system establishes a flexible yet efficient model of interprocess communication that allows incorporation of virtually any data transport mechanism. DUNE IPC extends the remote procedure call [Birrell & Nelson 1984; Spector 1982] model of interprocess communication. The traditional RPC mechanism is modeled on the standard procedure call found in almost all high level languages. Data and results are transmitted across a network at the beginning and end of the RPC transaction, respectively. The extensions we have provided address the following points.

- First, we are interested in providing complicated services, where the amount, source, and destination of the data

1. Other distributed systems that are built on top of general purpose operating systems like UNIX are not addressed in this paper. Such systems generally establish interprocess communication over the network facilities that the operating system provides and consequently do not permit the tailoring of communications for efficiency.

passing between the client and the server are unknown in advance. Thus the RPC mechanism must reach back across the network for additional data, as necessary. Examples of such complex services include process migration and the UNIX System V style of *ioctl* call.

- Second, we want to optimize the transfer of data as a function of the characteristics of differing communications channels. Breaking up large transfers into smaller packets may be unnecessary for some communications technologies. We want to retain the flexibility of allowing each interface to decide how it should handle the data. For example, if two processors can share a part of their address spaces, physically transferring data with a request for service is unnecessary; the data can be obtained directly when required at low cost.
- Third, with limited memory on individual processors the operating system may not have enough space to buffer large amounts of data accompanying a request. An example of this occurs during process migration in DUNE where an entire virtual address space is transmitted, but there is no movement of text or data until the system allocates physical memory for the remote process. Again, we require a mechanism where the data could be acquired as needed after sufficient space is allocated.

The DUNE IPC mechanism is called a *service request* and is used to exchange control flow and information between cooperating clients and servers. The service request uses an *access method* to support the optimized transfer of data. An access method exists for each medium supported by DUNE.

2. *The DUNE Distributed Operating System*

DUNE [Alberi & Pucci 1988; Pucci & Alberi 1988] is a distributed operating system using several forms of interconnect for communications. It is not a shared memory based system, although it can connect processors together using a conventional backplane. The

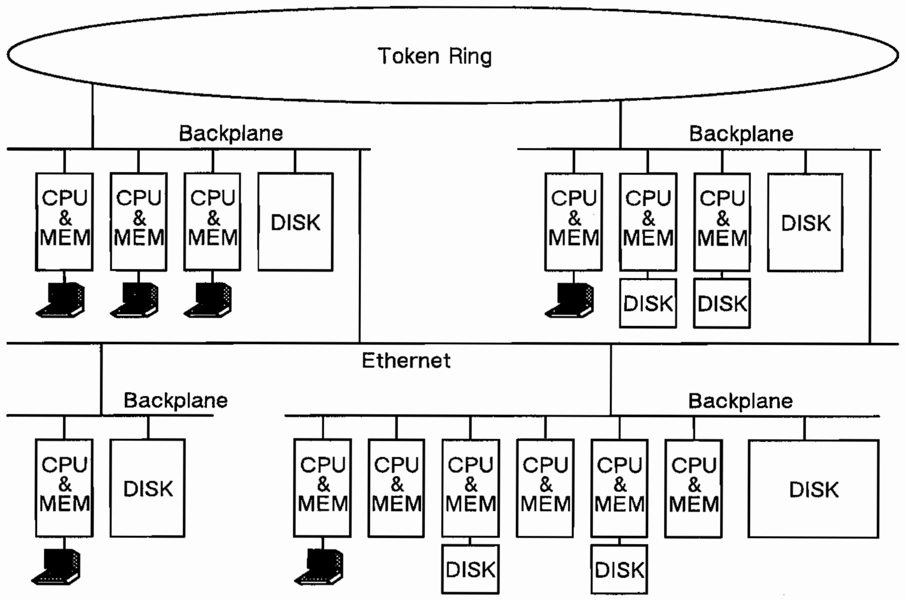


Figure 1: The current DUNE configuration. The Ethernet and token ring are connected in parallel to adapters resident in the backplane, which functions as another form of network.

current DUNE hardware configuration, shown in Figure 1, simultaneously supports service requests on the same processor, across the same backplane, across an Ethernet network, and across a high-speed token ring network. The DUNE system is composed of commercially available hardware. The single board computers are based on Motorola 68000 family processors with four megabytes of private memory. Up to eight such processors are connected on a single backplane, which is in turn connected to other backplanes via network interfaces.

While some of the media used by DUNE support broadcast or multicast operations in the hardware, not all do, and the conventional RPC model does not easily adapt to such features. There has been work done on RPC with multiple outstanding requests [Gifford & Glasser 1988], but we have not incorporated this feature into DUNE. The DUNE kernel simulates multicasts by iterating RPC calls through dynamic lists of resources that require the same operation. The kernel manages the lists as a function of the service being performed. Because simulated multicasts happen infrequently in DUNE, we conclude the marginal performance improvement is not worth the additional RPC complexity.

DUNE is a fully functional system with the semantics of the UNIX operating system [Ritchie & Thompson 1978]. The architecture we will discuss was instrumental in the smooth evolution of this system from its original form as a non-shared memory multiprocessor system into a distributed system. DUNE eliminates any perceived processor boundaries by distributing both the file system and processing space across all processors in the system. The system-call interface is enhanced from AT&T System V to include the Berkeley UNIX 4.2 network functions. The file system is singly rooted, hierarchical and independent of process location or user identity. A process can also read or write a physical device independent of location. Physical storage for the file system is scattered among the processors comprising the system. User processes are uniquely named throughout the system and may migrate between processors either automatically to balance load or by an explicit system call. Signals and process groups are fully developed in accordance with the semantics of UNIX System V.

3. *Software Architecture*

As in traditional UNIX systems, a user request for a system operation is initiated by a system call. Figure 2 shows the DUNE architecture; network connections are omitted for clarity. Horizontal lines represent interfaces between conceptual layers within the system. A user program does not issue service requests directly, but relies on the local kernel, hereinafter referred to as the client, to act as their agent for an operation. Control passes to the client via the usual system call trap. DUNE decomposes a UNIX system call into one or more service requests addressed to appropriate system resources, such as the root of the file system, the current working directory or the child processes of the current process.

The service request is the basic mechanism that distributes work either within the same processor or to other processors. The service request is similar to standard RPC but as a side effect conceptually transfers the entire client address space to the processor

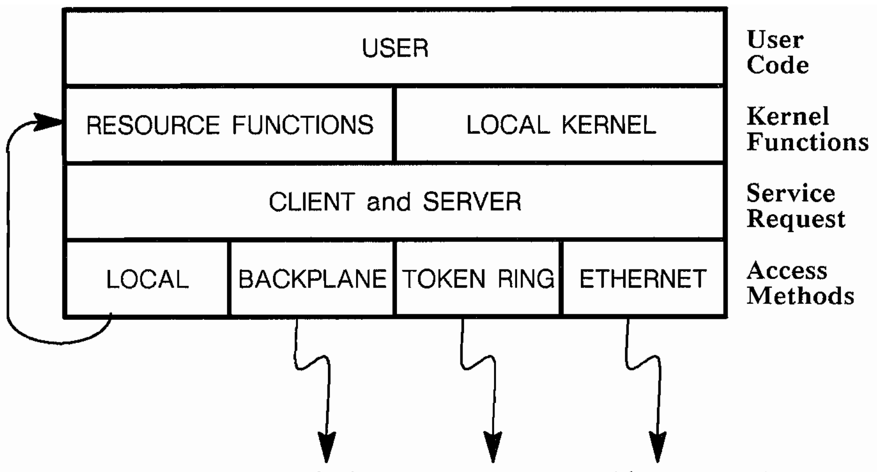


Figure 2: Layers within the DUNE RPC architecture. The system is symmetric with any processor acting as client or server.

containing the resource to be operated on by the request. This processor invokes local procedures in the remote kernel to operate directly (as far as it can tell) on any supplied or implied user data. The client determines the binding for the service, gathers arguments, constructs a hint describing the easily accessible data from the user program, transmits the request to the resource, and blocks awaiting the response. The request is queued at its destination and awaits dispatch by a server process responsible for the resource in the remote kernel. All servers are kernel processes and can directly issue additional service requests to complete a task.

The interface to each service request call includes a binding to the remote part of the request, outbound arguments (which can include arbitrarily indirect references to additional user data), a place to receive results, and optional *hints* to assist the underlying communications medium in optimizing data transfer. On the processor receiving the request, the provider of the service in the remote kernel has direct access to the explicit arguments and results of the call, as well as complete access to any indirect data references contained in the request. All access to the latter passes through the hint mechanism of the communications channel for device-specific optimizations. Each channel interface can also resolve general indirect references that are not satisfied by hinting, providing complete (although possibly less efficient) access to the entire caller's environment.

The kernel layer that supports the service request is termed an *access method* and incorporates a set of functions that deal with transport protocol and hint manipulation. We make no architectural distinction among network types, including the null network, and the requirements on the network protocol level are minimal. There is a null protocol for service requests to the same processor. All higher level functions, e.g. flow control, error recovery and connection management, are handled in end-to-end fashion at the level of the service request [Saltzer et al. 1981]. The transport functions may contain only error and duplicate detection or may be a complete end-to-end protocol. Although the latter case is redundant and causes performance degradation as will be shown later, general communications protocols may be desirable for other reasons, such as long haul transport.

An access method is a set of functions with a uniformly defined interface that supports the service request. Each access method is associated with a particular communications medium, and may include functions to send hints to the server about incoming data, move data between client and server, and send and receive messages. However, the access-method model does not require messages, and some access-method functions can be null operations. The high level kernel never deals directly with messages but only requests services from resources. Hint manipulation typically involves memory management changes that makes regions of address space available to other processors or physical devices.

Although the kernel is well structured, the boundaries between the layers in the kernel do not correspond to separate UNIX processes [Clark 1985]. For example, the client, which is part of the user's thread of control, executes functions from all levels including, in the case of the token ring, part of the network protocol. The efficiency of DUNE comes from this decoupling of processes and levels within the kernel. In the case of the TCP/IP protocol, performance is poor partly because the protocol is implemented as a separate process and partly because of the multiple layers in the protocol.

3.1 The Hint

The service request is different from a traditional remote procedure call because it has implicit access to the entire address space of the caller for both reading and writing. The hint reduces this requirement to more manageable dimensions which can be handled more efficiently. In practice, the entire address space of a client is not needed at the resource. A small but predictable region is sufficient, as long as unpredicted references can be accommodated. The client uses the hint to define the regions of data that it expects to be used by the resource. The access method uses the hint to give these regions preferential treatment.

Placing responsibility for the hint with the client rather than the server may appear nonintuitive. However, while complete knowledge of the data the server will access only resides with the server, by the time the server is activated, it is too late to optimize any transport. Hence, it is necessary to provide the client with some knowledge of what will occur at the server.

The hint can also add a degree of protection to a service request. As an option, the caller can restrict resource accesses to just the regions defined by the hint, thereby providing containment. This is possible since all server references to the client's data first pass through the hint mechanism to see if they have been optimized. If the security feature is enabled, any references not defined within the scope of the hint result in illegal access violations. In general, we have used this feature only for kernel-to-kernel operations to minimize inadvertent data corruption during testing of new features, and not for user processes.

3.2 Implementation of Service Requests

Permitting the lower level communications media to optimize certain data transfer operations justifies the added complexity to the RPC mechanism. The hint mechanism provides enough additional information to make such optimization possible, while not overly complicating the calling program. Since hints are optional, optimization can be added at a later time.

The basic structure of the service request appears in the client as:

```
     request (service, resource_link, arguments,
         results, hint)
```

Service is the desired operation to be performed on the resource identified by the *resource_link*. *Arguments* and *results* are the explicit inbound and outbound parameters for the operation and can be scalar or arbitrarily chained pointer references. *Hint* is a structure describing the information about the regions of the caller's address space that are expected to be used by the service request. For example, the hint for the process migration service request describes the type (outbound user data), address (segment origin) and size (segment length) for the text, data and stack regions of the process.

The virtual addresses described in the request are re-established in the server. Such information is crucial for the proper handling of unexpected data references or operations where complete semantics are unknown. Although it is possible to translate the buffer address implied in a *read* request to a local address appropriate to the server, it is not possible to remap flag arguments or arguments that can function as flags or pointers without knowing how these are to be used.

The service request is implemented on the client and server processors as cooperating pairs of medium-specific functions, which are summarized in Table 1. The behavior for each function is dependent on the particular communications device (if any) responsible for the connection. The *prologue* sets the stage for the subsequent request. For example, in the backplane access method described below, it is here that the memory management mappings of the buffers described by the hint will occur. If any data are to be pre-sent, as in the token ring access method, this will also occur here. The *request* transmits the actual message (if necessary) for queueing the desired operation at the server responsible for the resource.

During the actual use of client data at the server, a particular datum may or may not have been described by the hint mechanism. An expected datum reference will be satisfied on the server processor without any intervention on the client's processor. An unexpected datum reference will result in a delay as the server requests the datum from the client's processor. The client, which

Function	At Client	At Server
prologue	act upon hint	
		act upon hint
request	transmit request	
		await request, activate server
expected data request	(not involved)	access as though local
unexpected data request		request data from client
	perform desired access	
response		transmit results
	await results	
epilogue	tear down hint	
		tear down hint

Table 1: The structure of a service request.
Time increases in the downward direction.

is awaiting a response indicating it may proceed with its local execution, instead receives a request for access to its address space. It performs the access, returns the datum, and awaits the completion response or an additional request for data. The access, which is logically part of the service request, executes under the thread of control of the user process.

Once the request has been satisfied, the *epilogue* tears down the hint mechanisms established by the prologue. Any memory management mappings are invalidated, and pre-sent data are freed.

A further optimization occurs when the hint describes a small amount of data. Rather than using the prologue and epilogue functions for coordinating the data access, space for the data is allocated within the message used for the request and response. Currently, a hint composed of up to three defined regions and a total of 96 bytes of data can be accommodated. Under these circumstances, a remote service request consists of simply a request and a response message. The code used by the server to access client data conceals whether the data are attained through the message, the prologue/epilogue, or remote client access.

A 96 byte message buffer is chosen to accommodate most file path names and small I/O requests on terminals, thereby eliminating extra IPC transmissions for many common system calls. This value is also located close to the break-even point between the costs of copying data into a message versus adjusting memory management mappings to make the data available directly. This can be noticed as the small discontinuity in the performance chart for 96 byte backplane read operations shown in Figure 3 (in §4, below).

3.3 Local Service Requests

We are extremely concerned with the performance of the IPC mechanism in the degenerate case – when requests are satisfied on the same processor. The studies in Bershad et al. [1989] show that a large number of requests in a distributed system can be directed at local resources. For the sake of uniformity, we require that local operations continue to use the same IPC interface as truly remote operations – we do not want to litter the client with occurrences of:

```
if ( local )
   optimized code;
else
   use IPC interface;
```

as such usage is clumsy and error prone. Neither can one use early binding to convert general service requests to local function calls at build time because resources can move dynamically between processors, thereby altering the linkage between clients and servers. Hence, if a process migrates to the processor containing a file it is using, future disk accesses should bypass any real IPC operations.

Local calls are optimized in two ways: extra process switches are avoided by borrowing the context of the client, and the use of messages for a request is replaced by procedure calls. Since direct access to resources is faster than using any form of IPC, the access method functions dealing with hint manipulation are null procedures, and the request function calls the server function directly. This is possible because the request mechanism is synchronous, i.e., a requester suspends execution until the result is obtained. Under these circumstances, it is possible to use the flow of control of the requesting process to satisfy its own service request, and avoid two process switches between the client and a separate server.

The timings for various levels of optimization, shown in Table 2, justify our concern for local access efficiency. Data were obtained by constructing additional access methods with the specified characteristics (e.g., queueing messages, fielding interrupts, etc.) and measuring several thousand iterations of a simple system service (*seek*). The first entry corresponds to complete optimization – no messages are used and no context switches occur. These measurements also include user-to-kernel system call overhead.

Formatting, queueing, dequeueing and unformatting messages (one for the request, one for the response) add 330 μs. Most of this time is spent in packaging the parameters that identify the requester (user id, quotas, etc.).

The next entry includes the overhead of using separate server processes (as would be necessary if requests were non-blocking). Each process switch (client to server for request, server to client for response) adds another 150 μs.

Finally, if interrupts are used to deliver local messages, thereby fully mimicking the remote style of IPC, latency for the two interrupts involved adds another 330 μs. This brings the local time in close agreement with the communications cost of the shared memory (backplane) medium shown in the last entry. The small difference between the backplane time and that for the local case with server and interrupts is attributed to two factors that almost cancel each other: 1) There is greater expense in generating an interprocessor interrupt than an internally generated programmed interrupt; 2) Even though requests are synchronous, there is a

Condition	Time (in μs)
Local, fully optimized request	210
Above + formatting and queueing messages	540
Above + separate server process	840
Above + using interrupts for local delivery	1170
Remote backplane request	1200

Table 2: Round trip request times and the costs of layering

small degree of parallelism as a request suspends on the local processor while the service begins on the remote processor.

3.4 Backplane Service Requests

Although multiple processors connected through a backplane have byte-random access to common memory, DUNE uses the backplane only as a communications device – processes do not share data in an unconstrained manner across processor boundaries. Backplane service requests can be optimized by taking advantage of the memory management features of the processors, avoiding any data copying. The private memory of a processor can be dynamically mapped onto the address space of the backplane where it can be accessed directly by other processors that share the bus. Under these conditions, data movement is instantaneous and error free.

Unlike local requests, messages are now necessary to package and queue the requests and responses that cross processor boundaries. Backplane messages are allocated from a pool of memory common to all of the processors. Actual message transmission merely links the address of a message onto the receiver's queue – the message is not copied. The receiving processor is notified by a mailbox interrupt that schedules a server to process the request.

The movement of any data associated with a request or response is averted by the memory management mappings. The hint mechanism provides the backplane access method with the location and size of any data regions necessary for an operation. The client's prologue maps these regions onto the backplane's physical address space, making them externally accessible, and includes these mapped addresses in a part of the request message. The server's prologue uses both sets of addresses to alter the server's virtual address space. The address ranges described by the client's hint are re-established in the server but refer to the backplane addresses that were set up by the client's prologue. Hence, the original client data are directly available to the server at the same virtual addresses and as conventional program references.

3.5 Network Service Requests

Interprocess communication over a general network also uses the service request to optimize data transfers. DUNE has two networks that support service requests, an Ethernet and an 80 megabit token ring.

The protocol for the 80 megabit token ring is built into the functions that make up its access method. Therefore the protocol layer is distributed among the client process, the server process, and the interrupt code. The hardware handles low level acknowledgements and error detection. Because of the possibility of data errors, the protocol protects against duplicate packets. Higher level protocols, which apply to all access method, are contained in the service request and in kernel functions residing above the service request.

The TCP/IP protocol on the Ethernet is implemented in two separate processes attached to the access method functions by specialized message queues. Therefore activation of this protocol implies a context switch, which is costly even at the kernel level, and the management of yet another set of message queues.

The service request model can reduce the number of messages needed for an operation. Data small enough to fit in the request are included in the message; data too large are sent to the server before the request. This action implements the access method hint for these two networks. Because the server that receives a write request does not allocate buffer space for the data until it begins processing, the pre-sent data are tagged and temporarily sequestered by the kernel. The tag allows a server to retrieve the information locally and avoid the expense of more messages over the network. The data are retained on the server until the associated service request completes. Under heavy loads, the server may have no space for the data contained in a hint. The hint is then discarded, and the server must negotiate with the client via the network to retransmit the data at the appropriate time.

Retransmitting hinted data that is uncachable in the server is fundamentally different from returning to the client for unanticipated data. The prologue of the service request maps the hinted data to be visible to the network interface. The direct memory access feature built into both network interfaces allows efficient retransmission of the data. For the token ring, requeuing the transmission of the hinted data occurs at the interrupt level without interacting with the process that issued the original service request. For TCP/IP on the Ethernet, retransmission of the data requires some message passing and a context switch.

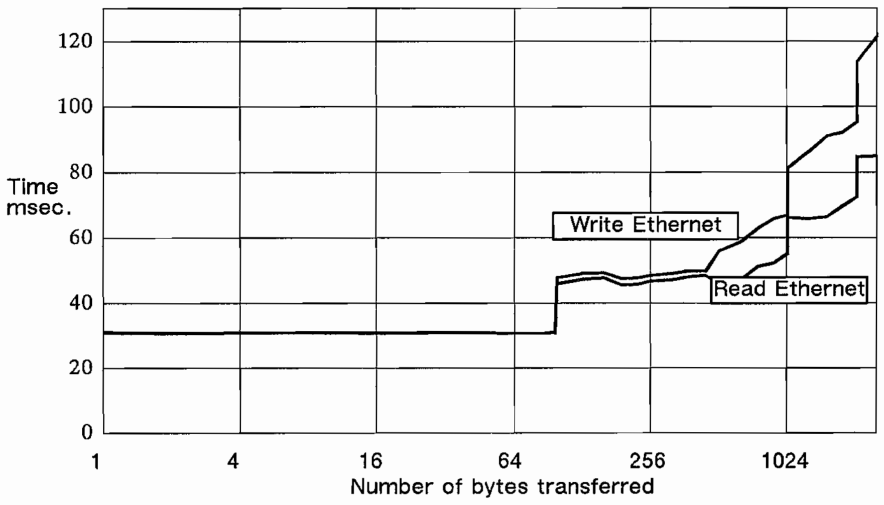
Within the architectural framework of the service request, the network hint and shared memory hint are identical even though they are operationally different. The shared memory hint is a mapping function of the processors' memory management units while the network hint requires sending and receiving messages. The local service request bypasses hints entirely. The hint mechanism is generally broad enough to improve the efficiency of almost any network.

4. *Performance Analysis*

The previous sections have described the IPC architecture for DUNE. The following is a set of measurements under controlled conditions that illustrate the system performance.

A test driver is installed in DUNE for demonstrating the improvements to data transfer through optimized access methods. The test device transfers an arbitrary amount of data, which are not interpreted, between user space and a 1024-byte kernel buffer. By eliminating all mechanical delays that arise from a real device and by varying only the access methods over the tests, measurements from the test device indicate the penalty imposed by each different access method and its associated interconnect. Measurements depicted in Figure 3 show the time to transfer a block of data to or from the test device versus the amount of data transferred. The abscissa is the number of bytes either read or written, and the ordinate is the amount of time needed to make the transfer. All measurements were taken on 68020 processor modules running at 20 MHz.

Several features evident in Figure 3 are the result of the performance improvements described above. The line labeled "Read Local" shows the transfer time for reading through the local access method. The write time in this case is not plotted as it is almost



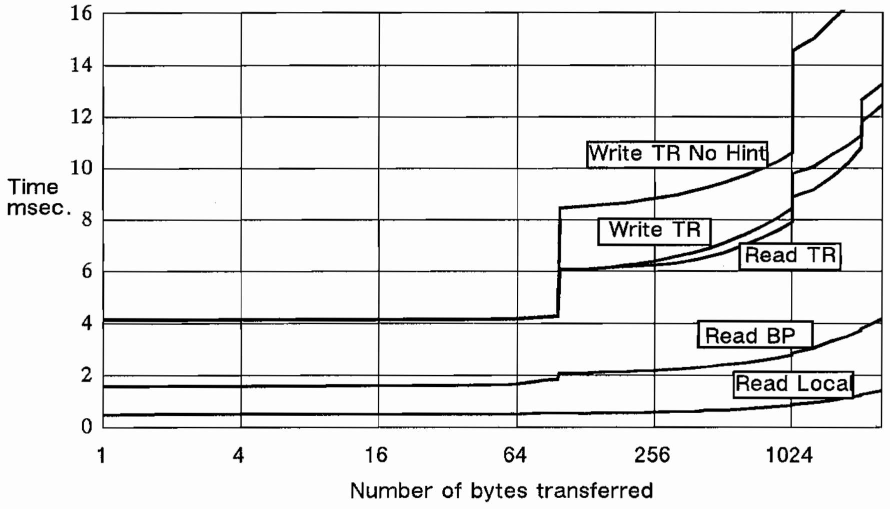


Figure 3: Transfer time as a function of data size for several communication fabrics and optimization levels. The lower graph illustrates (from bottom to top) reading from a local processor, reading across a backplane, reading across a token ring, writing across a token ring, and writing across a token ring with hints disabled. The upper graph shows reading and writing across the Ethernet using the TCP/IP protocol. Note the change in scale.

identical. The read time for one byte is 0.51 ms, and the time increases linearly thereafter at the rate of 0.38 μs/byte.

The penalty for having a remote server is illustrated in the plot labeled "Read BP." Because the read and write times are similar, only the read time is shown. The single byte transfer time is 1.6

ms. There is a step increase in transfer time at 96 bytes when the data can no longer piggyback on the service request message. The increased processing adds 0.5 ms to the transfer time at this point.

The two plots labeled "Read TR" and "Write TR" illustrate data transfer via the token ring. The basic 4.2 ms read or write time for a single byte of data reflects the network protocol processing and latency. The increased processing and message time is also evident when the data size reaches 96 bytes, but the equality of the read and write times at this point illustrates the lack of extra messages to fetch the data on writes. The subsequent divergence of read and write times arises when the server copies the pre-sent data to its final destination under program control. The divergence continues smoothly until the discontinuities at 1024 and 2048 bytes where multiple data messages are sent.

The physical characteristics of the network determine the need for multiple data messages for large data transfers. Because code for the network transport protocol resides in the access method, networks that accommodate arbitrarily large packets size will not cause these discontinuities. The backplane does not exhibit these discontinuities because data transfer is via memory management hardware on the individual processors.

The plot labeled "Write TR No Hint" demonstrates the effect of the hint contained in the access method. Each discontinuity is larger, which reflects the extra message traffic necessary to fetch the data across the network.

The two plots labeled "Read Ethernet" and "Write Ethernet" have features similar to those of the token ring except for the degraded performance. A single byte of data is written or read in 31 ms, and the break in the data at 96 bytes transferred is also evident. There are also the same discontinuities at 1024 and 2048 bytes, indicating transmission of multiple messages. The minor structure between these discontinuities arises from the fragmentation of messages and data into IP packets within the TCP/IP protocol.

Service requests over the Ethernet are measured to be approximately seven times slower than those over the token ring. We attribute this to several factors, including the complexity of the functionality in the general purpose protocol and the additional processes used in its implementation. Additionally, the inflexibility of the hardware device interface used in our current configuration compounds the performance problem. We have found results similar to those reported in Johnston & Campbell [1989] where the network interface can be excessively slow.

In our present implementation, message size and the point at which data are sent in a separate message are system wide constants. These values should perhaps depend on the individual access method invoked. The value that is best for the backplane access method might not be appropriate for other methods. The use of varying sized message buffers needs to be examined more closely.

5. *Relationship to Other Work*

The service-request paradigm is an extension of the semantics of the remote procedure call mechanism. The changes we have made address our needs to adapt to different underlying interconnects and handle complex operations. We have found it particularly useful to move information between client and server in an unstructured fashion. Supporting complex operations improves performance by reducing the number of interactions across the network that are required for a high-level user request.

While many systems have used RPC-like mechanisms for interprocess communications, Accent [Fitzgerald & Rashid 1986], Mach [Accetta et al. 1986], the V kernel [Cheriton 1984], and the x-kernel [Hutchinson et al. 1989] have features comparable to ours. Both Mach and Accent use features of the processor's memory management hardware in uniprocessor and multiprocessor systems to pass messages and to avoid the overhead of copying data between processes. The access method for backplane interconnections works in much the same way. Connections for transferring data are made by mapping areas of memory between processes. Partial and random access may then be made to the data. The V kernel, which is a network based system, can attach data to a request message although it is not clear to us that its protocol-driven mechanism can accommodate unforeseen data transfers.

Mach, Accent and V are all message-based systems, using explicit send and receive primitives for communication. While DUNE uses messages within some access methods, they are neither evident to higher levels nor a part of the structure of the system. Significant performance improvements are possible with local operations when the formatting and queueing of messages is avoided.

The access method for the local-area networks can both pre-send data and transfer any unforeseen items in the client process's address space. Mach's *network servers*, implemented as user processes, perform interprocess communication over the network. They correspond closely to the Ethernet access method and, although general and flexible, may suffer from non-optimal performance due to the extra layering and context switches. The Ethernet-based RPC described in Birrell & Nelson [1984] also bypasses the standard layers for a local network, but uses the protocol hierarchy for internetworking.

The x-kernel is designed to aid the construction of network protocols. As such it is not oriented towards general purpose computing like DUNE is. While not specifically oriented to RPC, it is like DUNE in supporting a common interface to all its protocols and run-time switching among protocols as a function of the interconnection. Protocols are stackable, and decomposable so that RPC can be built on a series of primitive layers. Unlike DUNE, however, both the sending and receiving protocol stack is encapsulated in a shepherding process. In DUNE parts of access methods execute at both process and interrupt levels, thereby saving the time to dispatch a process.

We have extracted features from the above systems and encapsulated them into extensible access methods that keep the details of the communication network hidden from the upper layers of the system, while providing increased efficiency.

6. Conclusions

This paper describes the service request, which encourages specific optimizations for particular devices while providing a simple and consistent interface to the higher level kernel functions. This mechanism is adaptable to nonconventional forms of media; it handles the degenerate case of local communications as well as incorporating shared memory backplane accesses into the communications model.

The networks supported by DUNE give some insight into the service request architecture as well as the implications of using special and general purpose protocols. The access method for each communications fabric is implemented differently even though functionally they are essentially the same.

The null access method for local service requests preserves the uniformity of the architecture while not compromising the efficiency of the system. This has proven to be quite useful, especially when the automatic load balancing features of DUNE dynamically rebind remote connections into local ones. The backplane access method demonstrates the flexibility of the service request architecture by efficiently treating a nontraditional interconnect as a communications network. The protocol for the 80 megabit token ring gains efficiency by eliminating additional processes and collapsing layers within the system. In contrast, the use of a layered protocol for the Ethernet permits it to function on long haul networks but at the cost of decreased performance.

We conclude from our experiences with networks and protocols that coalescing layers and moving end-to-end functionality to a high level directly improve system performance. DUNE demonstrates that a distributed operating system based on these principles can run efficiently on both multiprocessor and distributed hardware by including device specific optimizations for disparate networks.

References

M. J. Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian and Micheal Young, "Mach: A New Kernel Foundation For UNIX Development," *Proceedings of the Summer USENIX Conference*, pages 93-112, July 1986.

J. L. Alberi and M. F. Pucci, "The DUNE distributed operating system," *Proceedings of the First Using National Conference*, Denver, CO, September 1988. Also available as a Bellcore Technical Report, 1987.

Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, "Lightweight Remote Procedure Call," *Operating Systems Review*, (23)5:102-113, December 1989.

A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, (2)1:39-59, February 1984.

D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems," *IEEE Software*, 1(1):19-42, April 1984.

D. D. Clark, "The structuring of systems using upcalls," *Operating Systems Review*, (19)5:171-180, December 1985.

[DOD] "DOD standard transmission control protocol," *RFC-761*, Information Sciences Institute, Marina del Rey, CA, January 1980.

R. Fitzgerald and R. Rashid, "The integration of virtual memory management and interprocess communication in Accent," *ACM Transactions on Computer Systems*, (4)2:147-177, May 1986.

David K. Gifford and Nathan Glasser, "Remote Pipes and Procedures for Efficient Distributed Communication," *ACM Transactions on Computer Systems*, (6)3:258-283, August 1988.

Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott and Sean O'Malley, "RPC in the x-Kernel: Evaluating New Design Techniques," *Operating Systems Review*, (23)5:91-101, December 1989.

[ISO] "Transport Protocol Specification," *ISO/TC 97/SC 16, N 1169*, International Organization for Standardization, June 1982.

G. M. Johnston and R. H. Campbell, "An object-oriented implementation of distributed virtual memory," *Proceedings of the First Workshop on Distributed and Multiprocessor Systems*, Fort Lauderdale, FL, pages 39-57, October 1989.

G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, and G. Thiel, "LOCUS: A network transparent, high reliability distributed system," *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 160-168, December 1981.

M. F. Pucci and J. L. Alberi, "Optimized communication in an extended remote procedure call model," *Computer Architecture News*, pages 37-44, September 1988. Also available as a Bellcore Technical Report, 1987.

D. Ritchie and K. Thompson, "The UNIX timesharing system," *Bell System Technical Journal*, (57)6 part 2, pages 1905-1930, July-August 1978.

J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design" *Proceedings of the Second International Conference on Distributed Computing Systems*, pages 509-512, April 1981.

A. Z. Spector, "Performing remote operations efficiently on a local computer network," *Communications of the ACM*, (25)4:246-260, April 1982.