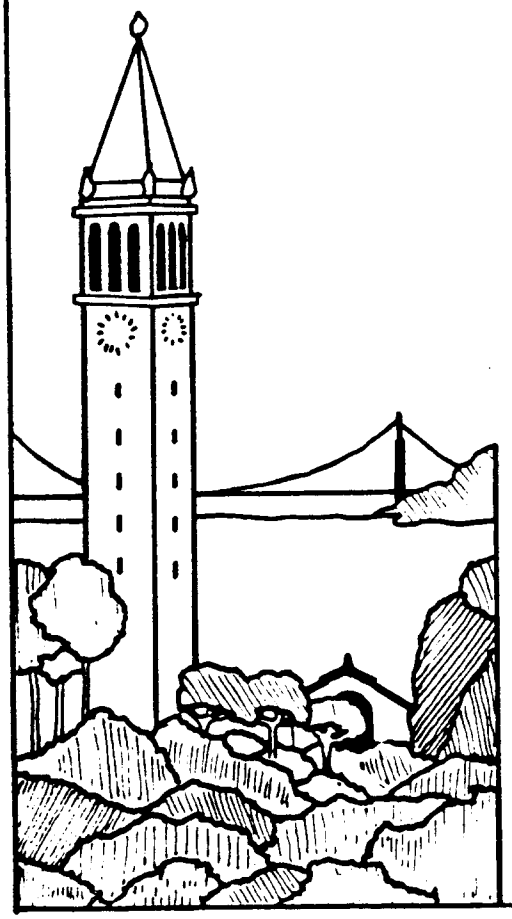
The Sprite Remote Procedure Call System

Brent B. Welch



Report No. UCB/CSD 86/302

June 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The Sprite Remote Procedure Call System

Brent B. Welch

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

This report describes Sprite's kernel-to-kernel remote procedure call (RPC) system. User programs access system services through the Sprite kernel on their local host, and the local kernel makes remote procedure calls to access services located on remote hosts. Each kernel has several RPC channels so that it can be making RPCs for several processes concurrently. A server machine keeps several kernel server processes that are used to execute service procedures. Messages carry requests for remote procedure execution to server kernels and carry results back to client kernels. The RPC network protocol requires only two messages per RPC in the common case. The protocol also supports efficient data transfers larger than the network packet size to meet the bandwidth requirements of Sprite's distributed filesystem. †

July 2, 1986

† This work was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-C-0269, in part by the National Science Foundation under grant ECS-8351961, and in part by General Motors Corporation.

1. Introduction

The Sprite operating system is designed for a set of co-operating hosts that communicate over a network. The details of network communication are hidden from users by modeling the Sprite system as a set of services available to user programs without regard to their network locations. Sprite services include a shared hierarchical filesystem [WeO85], remote program execution, and physical device access. User processes access all system services by making system calls into a local Sprite kernel. If a service is not implemented locally, the local Sprite kernel uses remote procedure call (RPC) to call a service procedure on a remote server machine. The Sprite RPC system lets the calling kernel, called the *client*, invoke the remote service procedure as if it were a local procedure.

The RPC system uses messages to carry requests for remote procedure execution to servers and to carry results back to clients. It uses a network protocol based on work by Birrell and Nelson [BiN84], which in the common case eliminates explicit acknowledgment messages; normally an RPC requires only a request message and a reply message. Eliminating extra messages reduces overhead; with only 2 messages per RPC, message handling still accounts for about half the overhead in each RPC. Performance results in Section 6 indicate 5.8 milliseconds of overhead for a remote procedure call between two Sun-2 workstations with Intel Ethernet controllers. The protocol also supports request and reply messages larger than the maximum network packet size. This allows high bandwidth communication, up to 380 Kbytes/second, for services like Sprite's network filesystem.

The structure of the RPC system is outlined in Section 2. Section 3 describes the RPC network protocol between a single client and a single server. Section 4 extends the protocol to a system of many clients and many servers. This includes allowing more than one outstanding RPC by client kernels, and allowing server kernels to execute more than one service procedure concurrently. Section 5 gives the low-level details of how messages are moved efficiently between the network and the processes using them. Section 6 gives the results of a few performance benchmarks and assesses the performance cost of different parts of the RPC system. Section 7 compares this RPC system with some others and Section 8 gives the conclusions of the report. Finally, there are a few appendices that fill in those details of the implementation that are not in the main report because they distract from the overall view of the system.

2. RPC System Structure

The standard model of RPC is that a remote service procedure is called by a client process on the local machine and executed by a server process on the

remote machine. In Sprite, the client and server processes are executing in their respective kernels' address space during an RPC. The client process is usually a user process that has entered the kernel during a system call. The server process is a kernel process, a process that lives entirely in the kernel's address space. The service procedures operate on kernel data structures so it is natural to use a kernel process to execute them. The client process can also be a kernel process. For example, on a diskless workstation the page-out process, which is a kernel process, is an RPC client when it writes dirty pages to a remote file server.

The bulk of the RPC system is a communication system used between the client process and the server process. Through the communication system, the client process issues a request for remote procedure execution, supplies input parameters to the remote procedure, and waits while the remote procedure executes. The server process uses the communication system to get the client's requests, and to return the results of the service procedure to the client process.

The heart of the communication system is the *network layer* that uses messages to communicate service requests and results between client and server processes. The network layer is described in detail in Section 3. The network layer's use of messages is hidden by a layer of stub procedures. The function of the stub layer is to package parameters and results into messages to be sent out, and to unpackage parameters and results from arriving messages. There are stub procedures on both the client and the server. Figure 1 illustrates the flow of control through the layers of the system during a remote procedure call.

The client stub procedure has the same parameters as the remote service procedure so the client process does not have to know that the service procedure executes remotely. The function of a client stub procedure is to package up the input parameters of the remote procedure so that the network layer can send them to the server process in a *request message*. After the request message is sent, the client stub waits for a *reply message* that the network layer returns to it from the server process. The reply message contains the results of the service procedure, and the client stub procedure unpacks those results from the reply message so its caller can access the results normally.

Server stubs are used by the server process so that service procedures do not have to handle messages. The server process gets request messages from the network layer and calls the server stub for the service procedure with the message data as a parameter. The function of a server stub procedure is to unpackage the input parameters from the request message, call the service procedure with its regular parameters, and package up the results of the service procedure. The server process then takes the packaged results and uses the network layer to return them to the client process.

The stubs for a service procedure can be generated automatically, and some RPC systems include stub compilers (eg. [BiN84]). This is useful in user-level RPC systems that are used in a large number of different programs. The Sprite RPC system, however, does not have a stub compiler. Although one would be a

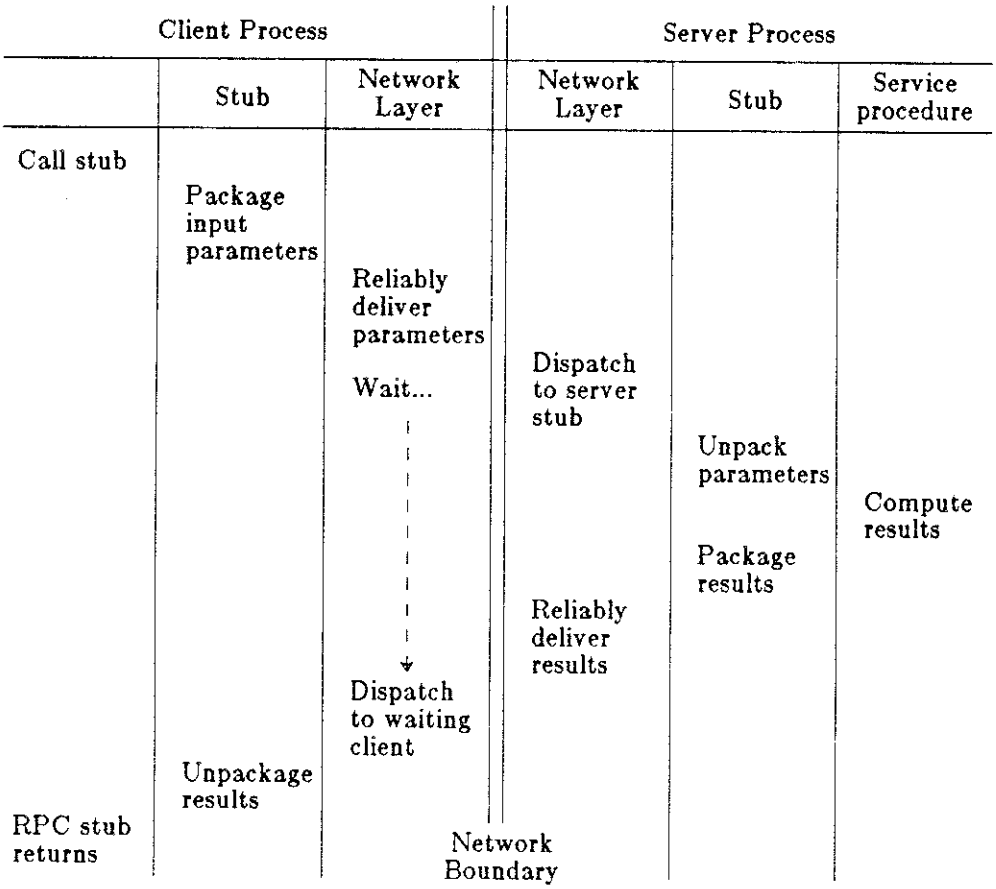


Figure 1. The execution thread as a client process calls a remote procedure. The client process blocks while the execution thread continues in the server process. The network layer is used to send messages between the client and server processes, and the stub procedures hide the network layer from the client process and the service procedure.

useful development tool, the effort of developing a stub compiler is greater than hand-coding the required stub procedures. Sprite only has a limited number of service procedures because it has a fixed set of distributed services. The Sprite filesystem, for example, is the biggest service, and it only has about 20 service procedures.

3. The Network Layer

The network layer implements the exchange of the request and reply messages between the the client process and the server process. Sprite uses a network, an Ethernet [MeB76], that limits the size of network packets and does not guarantee their delivery. To get a reliable exchange of request and reply messages, the RPC system uses a special-purpose network protocol; it requires only 2 network packets per RPC in the common case. The network protocol also

supports messages larger than the maximum network packet size so that service procedures can take or return large data blocks. This section describes the network protocol in a simplified world of only one client and one server, and Section 4 extends the RPC system to allow many clients and many servers.

3.1. The RPC Network Protocol

The RPC network protocol is designed to operate efficiently when there is a series of remote procedure calls between a client and the same server. Successive request and reply messages are used to acknowledge that previous messages were successfully transmitted. The receipt of a reply message by the client process acknowledges the safe arrival of the corresponding request message it sent to the server, and the receipt of a request message by the server process acknowledges the safe arrival of the previous reply message it sent to the client. This technique, called *implicit acknowledgment*, comes from the RPC work done by Birrell and Nelson [BiN84]. If no messages are lost, each RPC requires only the request message and the reply message; no extra acknowledgment messages are needed. See

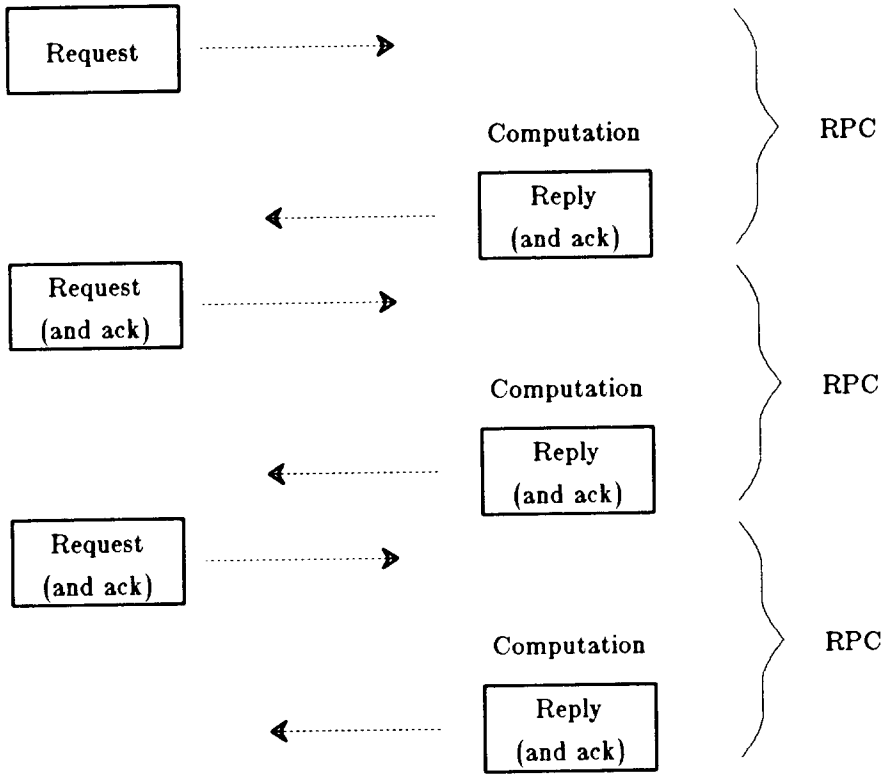


Figure 2. The RPC network protocol uses implicit acknowledgment to eliminate explicit acknowledgment messages. The server's reply message acknowledges the receipt of the client's request message, and the client's next request message acknowledges the receipt of the server's previous reply message.

Figure 2.

The implicit acknowledgment scheme needs a back-up in case a message gets lost. To guard against lost messages, a client process re-sends its request message if it does not get the reply message after a short timeout period. The retransmitted request message includes a flag in the control section of the message that says the client needs an explicit acknowledgment for the request message. When the network layer on the server gets a retransmitted request message there are three cases:

(1) The original request message was lost. In this case, the network layer on the server treats the request in the same way as it would have treated the original request: it passes the request message to the server process to initiate execution of the service procedure. In addition, the network layer also returns an explicit acknowledgment message to the client process.

(2) No messages were lost, but the service procedure is still executing. In this case, the network layer just returns an explicit acknowledgment message. This case is illustrated in Figure 3.

(3) The service procedure has completed but the reply message was lost. The re-transmitted request message does not cause the service procedure to execute again. Instead, the network layer on the server resends a saved copy of the reply message. This in an important feature because it means that a service procedure executes at most one time for each RPC by a client. A service procedure that locks a file, for example, will fail if it is executed an extra

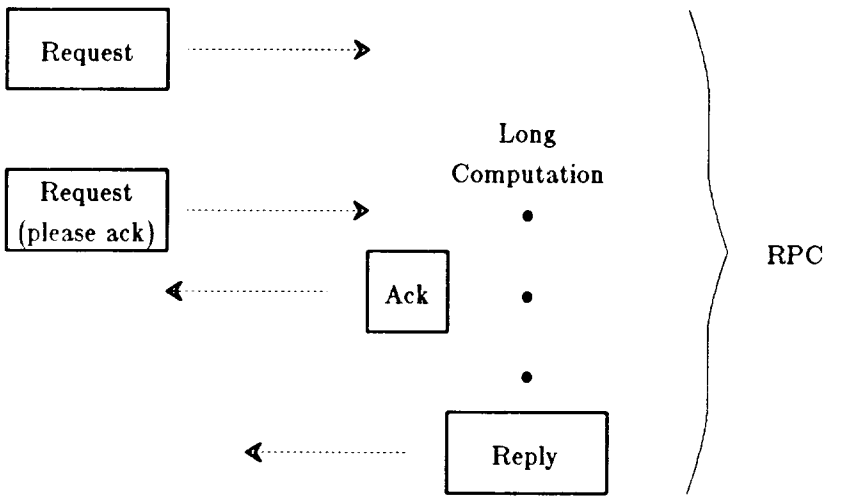


Figure 3. If the remote computation takes a long time the client process will get a timeout error before getting the reply message. In this case it re-sends its request message to the server process and requests an explicit acknowledgment. The network layer on the server responds with an explicit acknowledgment to let the client process know that the remote computation is in progress.

time.

If the server host is down then the client's re-sent request messages will not get explicitly acknowledged. After several retransmissions the client gives up and aborts the RPC. The server might also crash after sending an explicit acknowledgment, but before completing the service procedure and sending the reply. To detect this situation, the client continues to resend its request message at regular intervals, even after it has received an explicit acknowledgment. If the server host crashes while executing the service procedure the client process again gets several timeout errors and aborts the RPC.

The algorithm that the network layer uses in the client process to exchange the request and reply messages with the server is given in Figure 4. In the example, the Receive() procedure blocks the client process until a message arrives, or until the wait interval passes. Note that the request message is sent to a server

```
RPC_Call(serverID, requestMsg, replyMsgPtr)
    RpcHostID serverID;
    RpcMessage requestMsg, *replyMsgPtr;
{
    int wait, timeouts;

    wait = initialWaitPeriod;
    timeouts = 0;

    Send(requestMsg, serverID);

    while (TRUE) {
        status = Receive(replyMsgPtr, wait);
        if (status == TIMEOUT) {
            timeouts++;
            if (timeouts > maxTimeouts)
                return(TIMEOUT_ERROR);
            requestMsg.flags |= PLEASE_ACKNOWLEDGE;
            Send(requestMsg, serverID);
        } else if (status = ACKNOWLEDGMENT) {
            wait = wait * 2;
        } else {
            return(SUCCESS);
        }
    }
}
```

Figure 4. This is the algorithm the client process uses to send its request message to the server and wait for the reply message. The Receive() procedure blocks the client process until a message arrives, or until the wait period passes.

host, not to a particular server process; Section 4 discusses this issue in more detail. Also, each time an acknowledgment message is received the timeout period is doubled to give long-running service procedures a chance to complete without excessive overhead from retries.

3.2. RPC Sequence Numbers

The server kernel has a passive role in the network protocol; it leaves it up to the client process to make sure that it successfully gets the results of the service procedure. All that is needed on the server is a way to distinguish between new request messages and duplicate request messages sent when the client process retransmits. The server also needs to detect old messages if the network allows messages to get sidetracked so they arrive out of order.

The network layer on the server uses a sequence number in the control section of each message to determine if an incoming message applies to an old (invalid) RPC, the current RPC, or a new RPC. (Here it is assumed that there is only one client process so that there is only a single sequence of RPCs that the server has to monitor. This restriction is removed in Section 4.) The sequence numbers of all the messages associated with a particular RPC are the same, and the start of a new RPC is signaled by a message with a new sequence number. The client chooses the sequence numbers so they increase in successive RPCs. This allows the server to identify old (sidetracked) messages because their sequence numbers are lower than the current sequence number. The way clients initialize their sequence number when they boot is discussed in Appendix B.

3.3. Large Data Transfers

One of the main uses of the Sprite RPC system is the transfer of data blocks by Sprite's network filesystem. This sub-section describes how data blocks larger than the network's maximum packet size (about 1500 bytes with Ethernet) are transferred efficiently. The alternatives are to use several RPCs that each transfer an amount that can fit in one network packet, or to extend the network protocol so that more data can be transferred in a single RPC. Using several RPCs is less efficient because there is more overhead: each RPC has a fixed amount of overhead that comes from setting up the request message, waiting for the reply message, a process switch on both the client and the server, and other book-keeping like setting up timers. If the protocol is extended to be able to transfer more data in a single RPC, the fixed overhead is amortized over more data, and the transfer of a large block of data is faster. The improvement in performance by sending more data at one time has been presented in studies by Lazowska et. al. [LaZ84] and is also shown in the performance results given in Section 6.

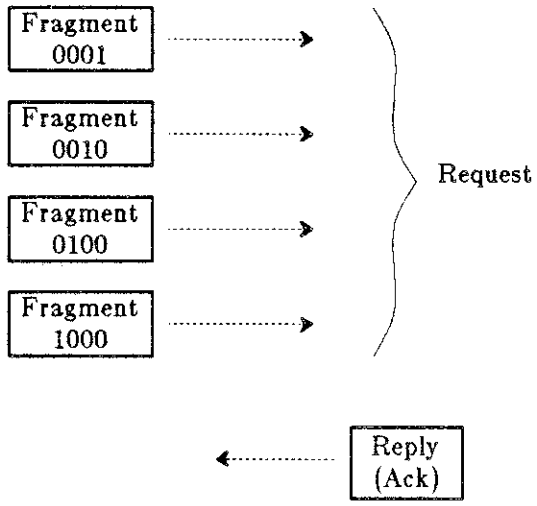


Figure 5. This shows a block transfer of 4 Kbytes of data in a single RPC. The request message is divided into 4 fragments that each contain 1 Kbyte of data. Each fragment is sent in its own network packet, and the I'th fragment is identified by a bitmask with the I'th bit set. All the fragments get sent before waiting for the reply message which acknowledges the receipt of all the fragments.

3.3.1. Message Fragments

Transferring a large data block in a single RPC is done by dividing large request and reply messages into *fragments*. Each fragment is sent in a network packet, and all the fragments are implicitly acknowledged by the next message from the receiver. Figure 5 illustrates fragmenting during an RPC that transfers a 4 Kbyte data block to the server. The request message gets divided into 4 fragments, and the server returns a reply after receiving all the fragments. Each fragment does not have to be acknowledged individually, and this contributes to the improvement in performance with fragmenting. The other main performance savings is that fragments are re-assembled into complete messages at interrupt time. This means there are the same number of process switches in an RPC that transfers a small amount of data and in one that uses large fragmented messages.

Each fragment contains four pieces of control information that are used during reassembly: the amount of data it contains, the offset of the data within the message, the number of fragments in the message, and a fragment identifier. The data size and offset allow the fragment to be copied into its correct location in the message's buffer when it arrives. This format also means that the loss of a fragment in the middle of a message does not prevent subsequent fragments from being copied correctly.

A fragment is identified by a bitmask: the I'th fragment of a message has the I'th bit of its identifying mask set. As fragments arrive, their identifying bitmask is or'd into a *summary bitmask* to record their arrival. The summary bitmask

indicates when the message is complete, and it detects if a fragment is a duplicate that arrived because of a retransmission. Duplicate fragments can be discarded without having to copy them into their destination buffer.

3.3.2. Lost Fragments

Lost message fragments raise a few issues, and the Sprite RPC system doesn't not yet solve all the problems in the best way. This sub-section describes the problems and the trade-offs, and Appendix C details the way the Sprite RPC system currently handles the situation.

The simple reaction to lost message fragments is to retransmit the entire message. If the receiver is a slower machine, however, this strategy may not work. This is because the fragments are sent out with almost no delay in between them so a slow receiver, or one with a poor network interface, may always drop the last fragments of a message. This problem is evident in Sun's network filesystem [NFS85] and RPC protocol [RPC85], and it is compounded because the protocol discards a message if any fragments are lost. Sun has two different machine types, and the newer Sun-3 is much faster than their Sun-2 model. The slower Sun-2 is not able to successfully receive a message that requires more than two network packets if it is sent from a Sun-3. To work around this problem, the Sun network filesystem limits the size of blocks sent from Sun-3's to Sun-2's so that at most two packets are used per message.

Another way to handle missing fragments, and the way the Sprite RPC system currently does it, is to optimize the re-send so that only the missing fragments are transmitted. This handles the situation of sending to a slow receiver, but it adds some complexity to the protocol. The receiver must tell the sender which fragments it has received. It does this by returning its summary bitmask to the sender when it sees that it has lost fragments; this is called a *partial acknowledgment*. The receiver also has to set up a timeout period after which it can check for missing fragments. While partial re-send will eventually get all the fragments to a poor receiver, it may require several iterations if the receiver can get only a couple fragments from each re-send. Also, as described in Appendix C, there are some messy details that crop up in the current implementation.

The final alternative is to re-send the message with a delay in between the output of each fragment. This gives extra time to a slow receiver, or to one that is just overloaded. The Sprite kernel has a *call-back* queue that is used to call procedures at specified times in the future. This can be used to set up calls to output the fragments at regular intervals. Also, delays could be used in conjunction with partial re-send. In this case, the first time a message is sent there would be no delay between fragments. Then, if some fragments were lost, the missing fragments would be re-sent with a delay in between them to increase the chance that the receiver gets them. The RPC system does not currently use delays, although they look promising. Future work includes experimentation with these ideas to determine a good solution to the problem of lost fragments.

4. Many Clients and Many Servers

The last section presented the RPC network protocol in terms of a single client process and a single server host with one server process. This section extends the RPC system to a world of many client and server hosts. This section also presents extensions that support concurrency within each host so that a client kernel can have more than one process making a remote procedure call at the same time, and so that a server kernel can service many clients at one time.

4.1. Client Channels

The Sprite RPC system uses *client channels* to connect processes on a client host to a server process on a server host. The use of channels reduces the state information that a server has to keep. A server does not have to know about all the processes which access the server, only about the client channels they use. Each host keeps a few client channels so that more than one process can make RPCs at the same time, and any process can use any free client channel. A simple system of two client hosts and one server host is shown in Figure 6. The effect of a client channel is to serialize the RPCs from the various processes on the client host that use the channel so that the network protocol can use implicit acknowledgment; the next request message from the client channel implicitly acknowledges the last reply message sent to the channel.

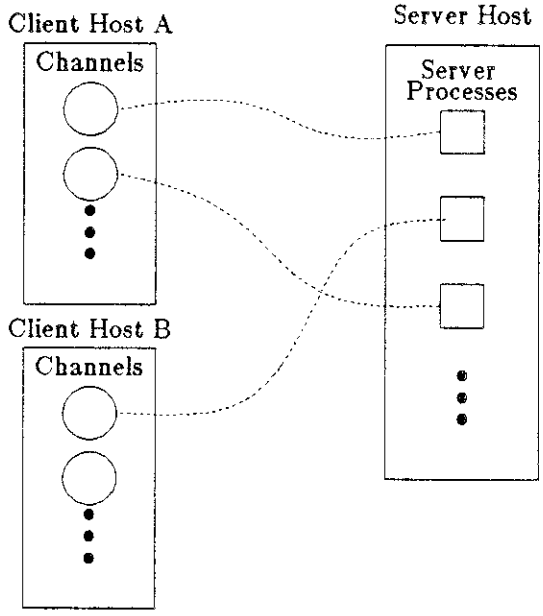


Figure 6. A server host keeps several server processes so it can handle RPC requests from more than one client. A client process is connected to a server process over a client channel, and each kernel has more than one client channel so more than one of its processes can do RPC at the same time. RPCs are serialized over a client channel so the network protocol can use implicit acknowledgment.

4.2. Multiple Server Processes

A server kernel keeps many server processes so that it can execute service procedures for many clients at one time. It is not practical, however, for a server host to have a different server process for every possible client channel in the system; Sprite is designed for an environment of a few hundred hosts and each one has several client channels. Instead, a server host keeps a limited number of server processes and multiplexes them between the client channels that use the server. This can be done because usually only a small number of client channels are using the server at any one time.

A server host assigns a server process to a particular client channel for a series of RPCs. When a request comes from a client channel that has no server process, the network layer on the server assigns a free server process to the client channel. The server process remains assigned to the client channel until the network layer on the server notices that it has not received a new request message from the channel recently. At that point, the network layer sends a probe message to the client channel that requests an explicit acknowledgment for the last reply to the channel. The explicit acknowledgment from the client channel ends the series of RPCs between it and the server process so the server process is free to be re-assigned to an active client channel.

A server only keeps track of the RPC protocol for those channels to which it has assigned a server process, and the state for the RPC protocol between a channel and a server process is kept in a data strucure associated with the server process. To help the network layer on the server match a message from a client channel with the correct server process, all messages from a client channel give the ID of the server process that was used to service the channel's last request. The ID is returned to the client with each reply and explicit acknowledgment so that it can provide the information for the next RPC. Most of the time the server process suggested by the client channel is correct. If the channel has been inactive, however, the server process has probably been re-assigned to another active client channel. If this is the case, the server searches its pool of server processes for a free one and assigns it to the client channel.

The use of just a few client channels by a client host, as opposed to a client channel for every process, for example, increases the effectiveness of assigning a server process to a client channel for a series of RPCs. The RPCs from all the processes on the client host are concentrated onto a few channels so the rate at which RPCs are done on each channel is more than the rate from each individual process. This reduces the amount of time the server process for the channel waits for a new request, and it also reduces the chance that the server process will get taken away and assigned to a more active client channel.

The network layer on the server checks periodically for server processes that are assigned to inactive channels. It schedules a scavenging procedure to be called about once a second that scans the pool of server processes for ones than have been idle since the last time it checked. If a server process has been idle then the

scavenger sends a probe message to the client channel.

The rate at which the server processes are checked by the scavenger has to be low enough so that it doesn't add too much overhead to the system, but is also has to be fast enough so that server processes can be reclaimed during times of peak load. It is important to have free server processes because the server discards requests if there is no free server process. This problem, client starvation, can be made rare by keeping enough server processes to handle peak loads. For example, a prototype of Sprite's network filesystem was implemented in a system of about 12 workstations, and the main file server kept 8 server processes. The system was in daily use for several months and no client requests were ever discarded. Still, more experience is needed with Sprite in a system with a large number of hosts to be able to tune the rate at which server processes are reclaimed and to determine how many server processes should be kept.

4.3. Broadcast RPC

The final addition to the RPC protocol for a real environment of many servers is a broadcast form of RPC. Broadcast RPC is implemented by broadcasting the request message to all hosts (Ethernet supports broadcasts), giving the first reply to the client process, and discarding any subsequent replies. All hosts have to process broadcast messages to some extent so broadcasts are expensive to the network community. Because of this, the RPC system does not retry if there is no reply after the timeout period. Instead, the client process is returned an error so that it can retry the RPC at a rate that does not saturate the network. These semantics of the broadcast RPC make it suitable for clients that need to locate server hosts. It is not a reliable broadcast, one that ensures that all hosts have received the request. That is rather expensive to implement and it is not required by any Sprite services.

5. Message Handling

The previous two sections have glossed over the details of how the network layer handles messages. This section describes how the network layer is organized to handle messages efficiently. The network layer is divided into the *network module* and the RPC *dispatcher*. The network module sends and receives packets over the network, and it is independent of the protocol sending the messages. The RPC dispatcher is called by the network module at interrupt time when a packet for the RPC protocol arrives.

The network layer does three things to transmit messages efficiently. The first is that, when possible, the network module uses the basic Ethernet protocol [MeB76] to send network packets between machines. The second is that part of the RPC network protocol is implemented at interrupt time by the RPC dispatcher so some messages can be handled without incurring the cost of a process switch into a client or server process. The third is that the buffering system that the network module and the RPC dispatcher communicate with is designed

to minimize copies in large block transfers.

The advantage of using the basic Ethernet protocol is that it is a bare-bones protocol that is efficient to implement. The disadvantage is that packets can only be transmitted over a single Ethernet cable. The RPC protocol does not depend on the protocol used by the network module, however. The network module uses the basic Ethernet protocol to reach hosts on the same ethernet, and more complex protocols to reach Sprite hosts on other networks. Currently the network module only implements the basic Ethernet protocol, but the Internet Protocol(IP) [IP80], a datagram protocol that handles packet routing in an internet, will be supported in the future.

5.1. The RPC Dispatcher

The RPC dispatcher is called at interrupt time by the network module to handle an arriving packet. The dispatcher is divided into the client dispatcher and the server dispatcher because the RPC protocol is asymmetric. Three general cases for both dispatchers are listed here, and the details of what happens in the client and server dispatcher are discussed below.

(1) The packet is a complete message needed by the receiving process. The dispatcher copies the packet into buffers owned by the receiving process and notifies the process that it has received a message.

(2) The packet is a message fragment. The dispatcher copies the fragment to its offset within the message buffer and notifies the receiver only if the message is complete.

(3) The packet is a message that is not needed by the receiving process. The dispatcher handles the message according to the RPC protocol without giving it to a process. This is done either because there is no process waiting for the message, or because it is more efficient for the dispatcher to handle the packet. This saves the cost of copying the message to the receiver's buffer and switching to the receiving process.

The client dispatcher passes replies and explicit acknowledgments to client processes without taking any special action. Other messages, however, are processed by the client dispatcher, and the client process (if any) is not notified that a message arrived. For example, the dispatcher handles probe messages that the server sends after the client channel becomes in-active. At this point there is no process using the channel, or it is being used for RPC with a different server, so the dispatcher returns the explicit acknowledgment to the server. The client dispatcher also handles partial acknowledgments as described in Appendix C.

The server dispatcher only passes request messages to server processes. The other kinds of messages — duplicate requests, probes, and acknowledgments — are handled by the dispatcher because the server process may be busy executing a service procedure. Even if the server process is not busy, handling these other kinds of messages in the dispatcher saves the cost of a process switch and

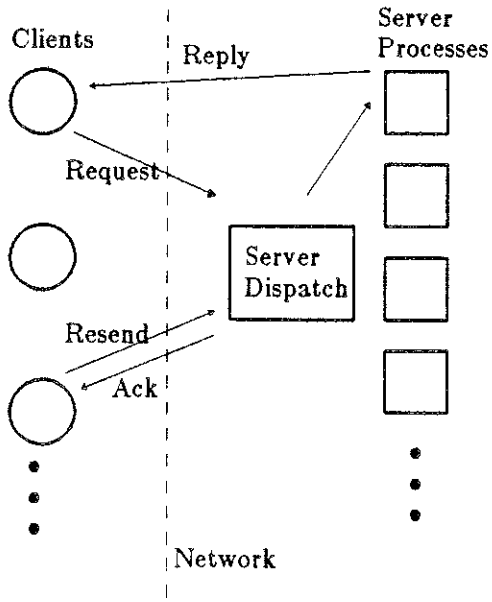


Figure 7. This figure depicts message traffic from the server's point of view. The server dispatcher passes new requests to server processes which later return reply messages to their clients. The dispatcher also takes care of returning explicit acknowledgments and re-sending replies that clients have lost.

simplifies the server processes. This means that the special cases in the RPC protocol on the server side (things like re-sending replies and returning explicit acknowledgments) are implemented by the server dispatcher instead of the server processes. See Figure 7.

Before the server dispatcher can take any action for the server process, however, it must decide which is the correct server process. This is because when a message comes from a client channel it might not specify the correct server process. It does include the ID of the server process that handled the previous request from the client channel, but as was described in Section 4.2 the server process may have been re-assigned to another client channel. It is the job of the server dispatcher to check that the server process is still assigned to the client channel, and to choose a new server process if necessary.

5.2. Building Messages

The RPC system supports the gradual assembly of a message as it is passed through the various layers in the implementation. A message starts forming in the stub procedures as they package up their parameters. A stub procedure then passes the message to the network layer which adds control information to the front of it. Finally, the network module adds low-level addressing information to the message for transmission on the network. One goal of the RPC system is to be able to build messages gradually without having to re-copy a message each

time information is added to it.

To allow a message to be built up gradually, the various pieces of a message are each stored in separate buffers, and the buffers are linked together in a list. See Figure 8. Stub procedures start creating a message by setting up a list of one buffer that contains the parameters in the message. The network layer adds control and addressing information to the front of the message by pre-pending buffers to the list. Finally, the network module uses a DMA Ethernet controller, the Intel 82586, that copies the data from the list of buffers directly to the network. This means data is copied only one time, from its buffer to the network.

The use of a buffer list also makes setting up a fragmented message easy. A buffer list for the fragment is set up that references only the part of the message data that goes into the fragment. This means that the data for a fragment does not have to be copied into a new buffer. The third buffer in Figure 8, for example, might really be a section of a much larger buffer.

5.3. Unpacking Messages

The counter-part to the problem of gradually forming messages is the problem of efficiently unpacking the parameters from a message. A simple approach ends up copying parameters three times: the network hardware copies a message once from the network to a buffer in the network module, the dispatcher copies

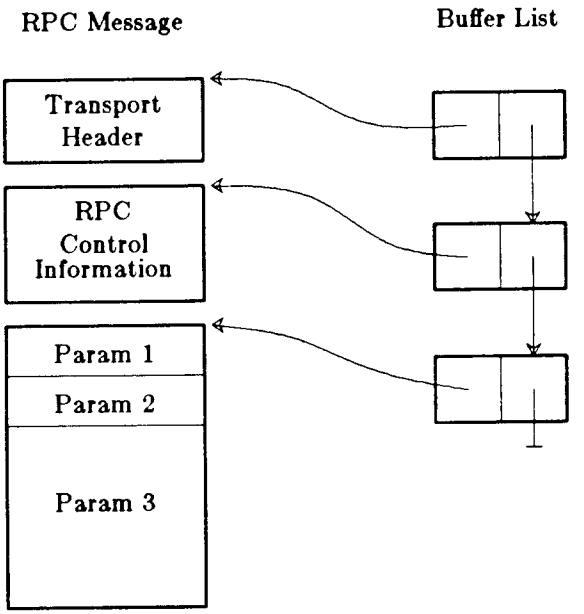


Figure 8. A message has three sections: the transport header, RPC control information, and the parameters of the service procedure. The sections are kept in separate buffers to they can be set up by different layers of the implementation. The network module uses a DMA interface that copies directly from the list of buffers to the network.

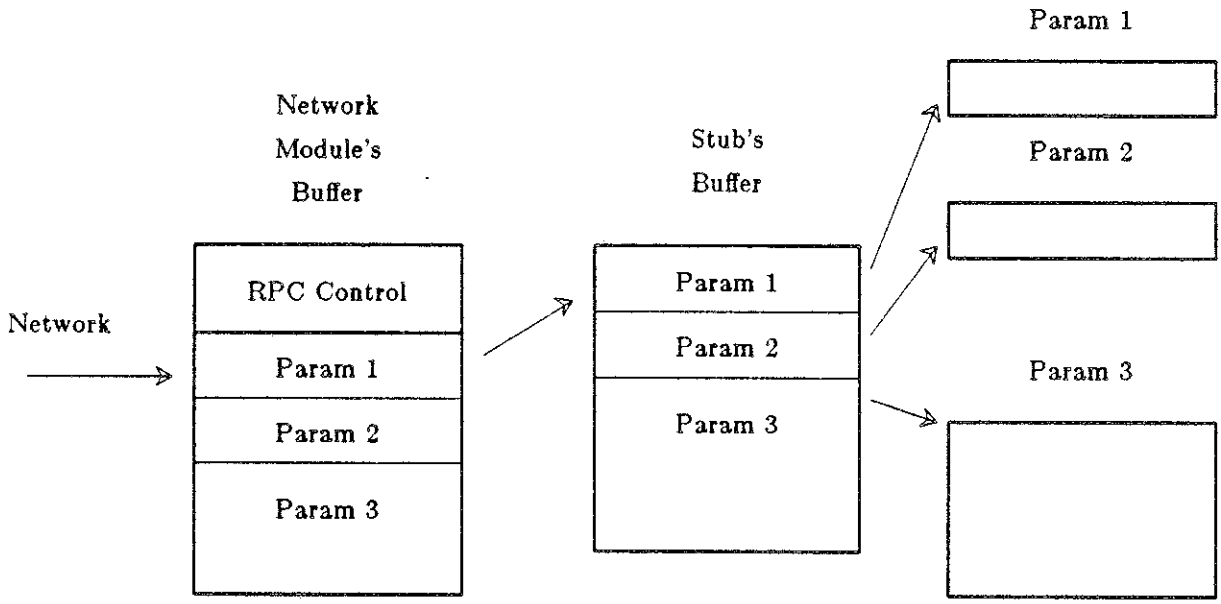


Figure 9. This shows a buffering system that requires copying the parameters in an arriving message three times, each arrow represents a copy.

the parameters in the message a second time to another buffer owned by the stub procedure, and the stub procedure copies each parameter a third time to the final location of the parameter. This is shown in Figure 9. Copy costs become important with large parameters; it takes about .8 milliseconds to copy 1 Kbyte on a Sun-2.

It is tempting to try have only a single copy by having the network hardware copy the parameters directly (via DMA) from the network to their final locations. This is not practical, however, because it requires that the system anticipate what kind of message will arrive next so it can set up buffers ahead of time, and it requires that the hardware interpret some control information in each message that specifies the size of each parameter. It is also tempting to pass the network module's buffer to the stub procedure in order to save a copy. This is not done, however, because the network module can save on setup costs by keeping a small number of buffers and recycling them quickly.

The RPC system takes an intermediate approach that saves a copy for the largest parameter in a message. This is an optimization oriented towards the block transfer requirements of the filesystem; the read and write service procedures have a few small parameters and a single large data block. The parameters in a message are divided into two sections, and the dispatcher copies the two sections to separate buffers. This allows the stub procedures to set things up so that the dispatcher copies the large data block in a message directly to its final location. This is illustrated in Figure 10.

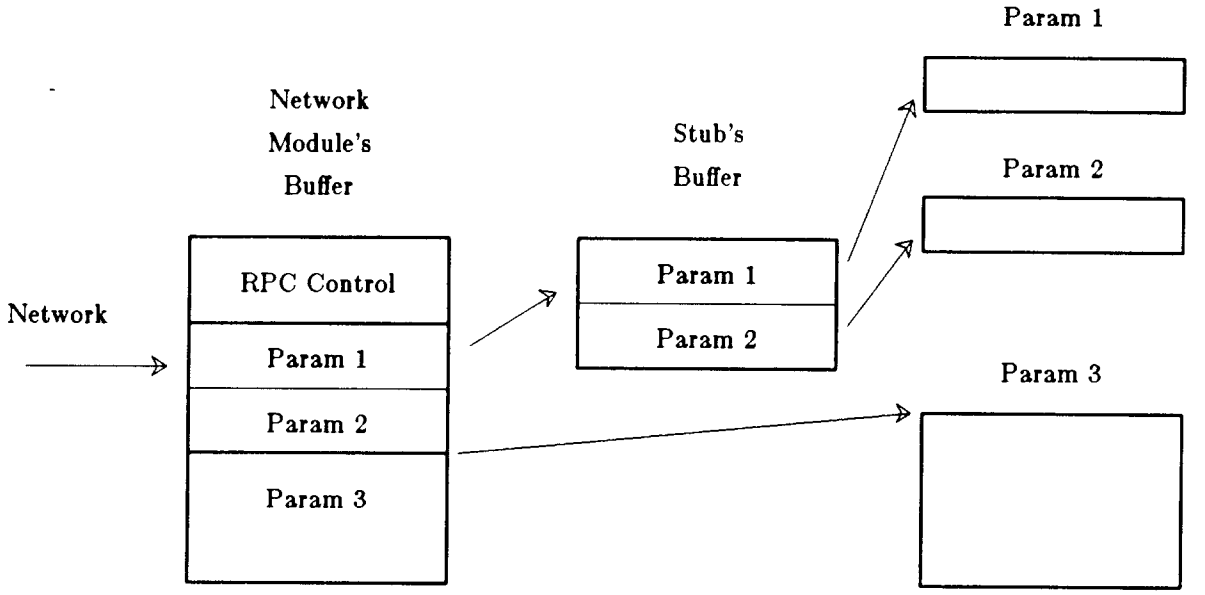


Figure 10. The parameters in a message are divided into two sections and the dispatcher copies the two sections to two separate buffers. The stubs put a large data block in the second section and set up the buffers so that the dispatcher copies the data block directly to its final location.

The parameters in messages being sent out are also divided into two sections, and this allows the same sort of optimization. For messages being sent out, a stub can set things up so a large data block is copied directly from its location to the network. This means that a large data block only gets copied 3 times as it is moved from the client process's buffer across the network to the server process's buffer; there is one copy on output, and there are two copies on input. Appendix D has a client and server stub procedure that illustrate the use of two data areas.

A more general version of this scheme would be to have a variable amount of control information in each message that specified the number and size of each parameter. This would allow the dispatcher to copy all the parameters to their final locations instead of copying some to an intermediate buffer. The RPC system uses a fixed format, however, because it simplifies the implementation and still optimizes the common case of one large parameter and several small ones.

Implicit in this discussion is that receivers have to set up buffers before the message arrives. Clients can do this easily because they know what to expect in a reply message. Servers, however, have to keep buffers ready that are large enough to handle the largest request message. The main advantage of pre-allocated buffers is that receivers can optimize the buffering for the largest parameter. Another benefit is that memory allocation (if any) is not in the critical path of message handling; in clients it is done before the RPC, and in a server process it is done after the reply message has been sent.

6. Performance

This section presents some benchmarks and assesses the cost of different parts of the RPC system. The benchmarks were executed on the Sprite operating system in spring of 1986. Sprite at that point supported multitasking, priority scheduling, user programs, virtual memory, and an incomplete (client side only) version of the filesystem. The kernel was still under development and was not yet used for day-to-day work.

The hardware used for the benchmarks were Sun-2's with Intel 82586 Ethernet controllers on a 10 Megabit Ethernet. The Intel controllers are very good and rarely lose packets. They are configured with 18 buffers for receiving packets so a packet is only lost if 19 or more packets arrive back-to-back. (Sprite also runs on machines with 3Com ethernet controllers, but they can only buffer two packets and were not used in the performance benchmarks.)

6.1. RPC Overhead and Bandwidth

The basic cost of RPC and the bandwidth of the system were measured by timing a sequence of Send RPCs. The Send remote procedure call just sends a block of data from the client to the server. The amount of data sent was varied from 0 to 14 Kbytes, and three runs were made at each different size. A send of zero bytes measures the fixed overhead in an RPC, and sizes greater than 1 Kbyte measure the effects of fragmenting.

The time to send different amounts of data is plotted in Figure 11. Each test was done three times so there are three points plotted at each size. Each time is the elapsed real time divided by the number of RPCs done; this averages in overhead from all layers of the implementation as well as overhead from other parts of the Sprite kernel like priority scheduling. The effect of fragmenting is evident in the steps that occur just past each multiple of 1 Kbyte. (Actually, messages up to about 1400 bytes can fit in one packet. Fragmenting is still done on 1 Kbyte boundaries, but fragmenting isn't turned on until a message won't fit in one packet. That's why the first step in the graph occurs past the 1 Kbyte point.) Other than the steps, the plot shows a nearly constant increase in the time for each additional 1 Kbyte of data sent, about 2.4 milliseconds/Kbyte. This slope indicates a limiting bandwidth of 415 Kbytes/sec as the size of the packets is increased. Saving a copy as described in Section 5.3 saves .8 msec/Kbyte. If this were not done, the slope of the graph in Figure 12 would be 3.2 msec/Kbyte, and the limiting bandwidth would only be 310 Kbytes/sec.

As plotted in Figure 12, the bandwidth starts to level out around 380 Kbytes/sec, which is about 30% of the Ethernet's 10 Mbit capacity. This supports Lazowska's result from tests on similar hardware that the server's CPU is the bottle-neck and the network is underutilized [LaZ84]. Even blocksizes of 4 Kbytes and 8 Kbytes result in good bandwidth, about 275 and 330 Kbytes/sec respectively. If the system was limited to 1 Kbyte messages, ie. fragmenting was not implemented, the maximum bandwidth would be about 135 Kbytes/sec. The

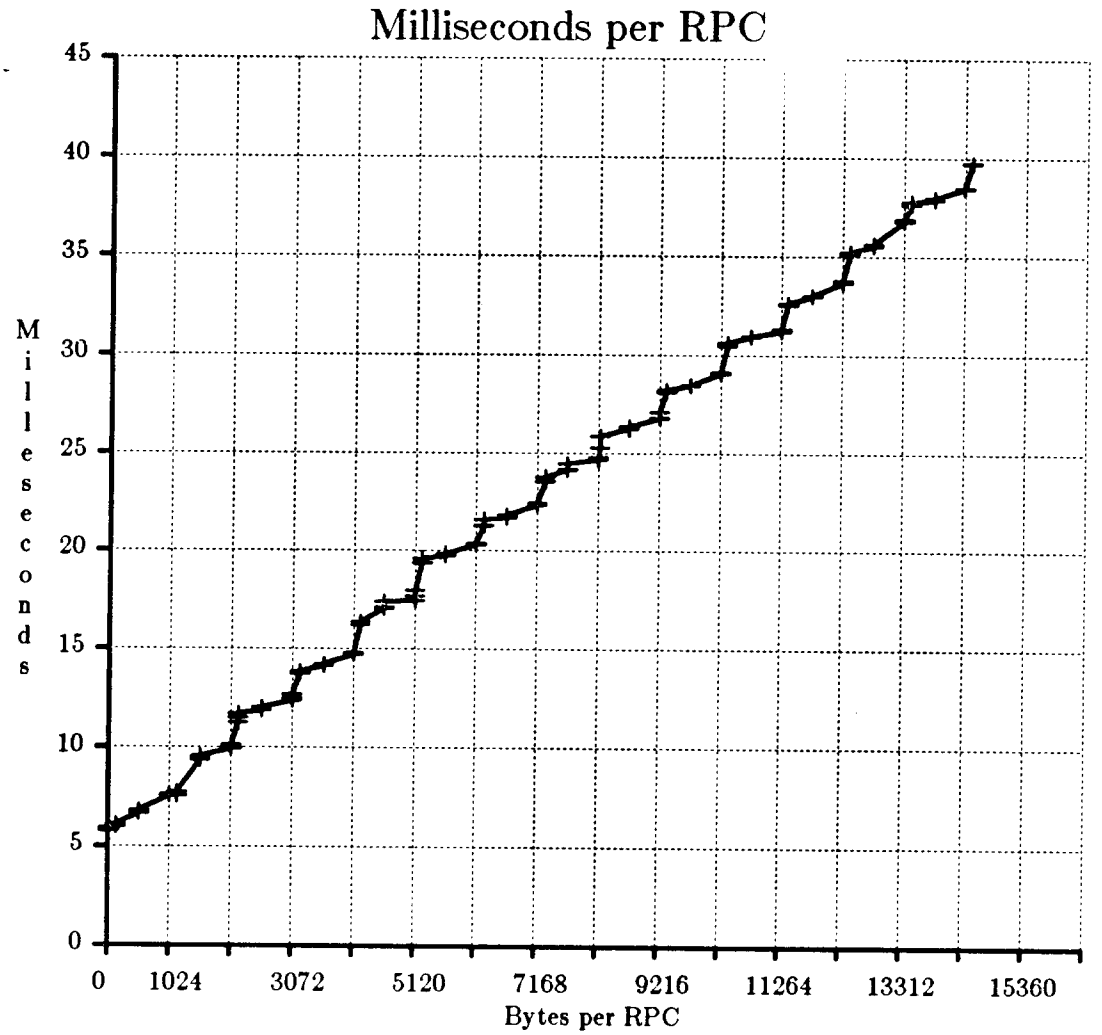


Figure 11. The time for a Send RPC with various amounts of data. Each test was done three times so there are three points plotted at each size. The times are the average of the real time required to perform a series of RPCs, including system overhead. The system fragments on 1 Kbyte boundaries; this is evident in the steps in the graph. The overall slope of the graph is 2.4 milliseconds/Kbyte. This indicates a limiting bandwidth of 415 Kbytes/sec.

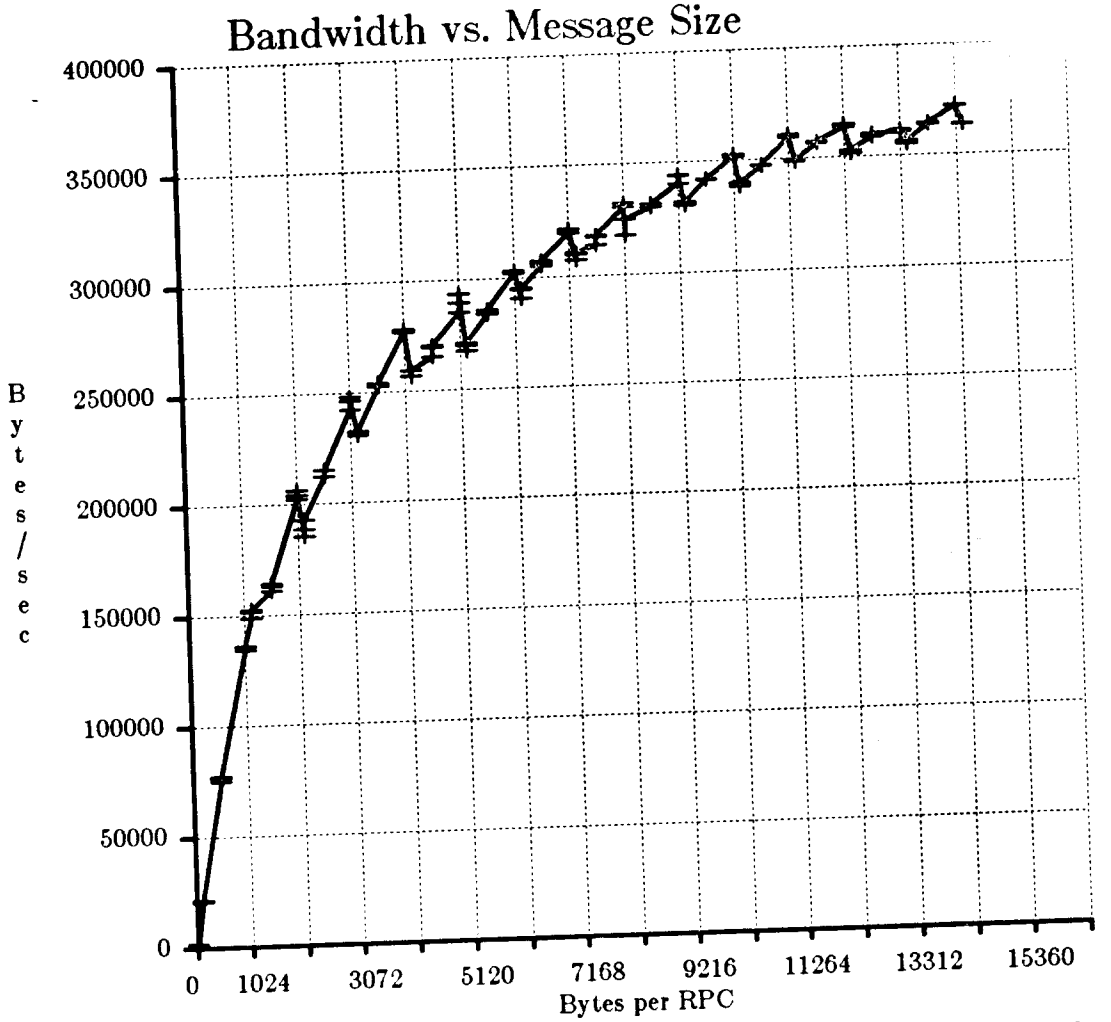


Figure 12. A plot of bandwidth vs. message size. The bandwidth increases with message size but it starts to level out around 380 Kbytes/sec. The notches in the bandwidth occur each time an additional fragment is needed to send the request message.

bandwidth from the Send RPC represents an upper bound on the bandwidth available to Sprite's network filesystem, and it is comparable to the effective disk bandwidths, 220 and 440 Kbytes/sec, reported by McKusick [McK84] for the 4.2BSD UNIX filesystem.

6.2. Comparison to the V system

The V system is a message passing system that has been tuned for high performance. Its performance is a lower bound on the overhead that the RPC system has from its use of messages. Table 1 lists some performance results from V [Che84] and the corresponding results for Sprite; both sets of results come from tests on Sun-2 workstations. As the null send operation indicates, there is a considerable amount of overhead in the Sprite implementation of RPC. About 2.8

Sprite vs. V system		
Operation	Sprite	V system
Null send	5.8	2.2
1024 byte send	7.5	5.8

Table 1. The performace of similar operations in the V-system and Sprite. About 2.8 msec of the 5.8 msec of Sprite overhead is due to message handling. The rest is from other parts of the RPC system.

msec of this overhead is directly related to message handling, and this close to the cost of sending a message and getting a reply in the V system. This means that the message handling as described in Section 5 is ok, but that other parts of Sprite need tuning.

6.3. Contributions to Overhead

A more detailed tracing was done to determine the time spent in various parts of the RPC system. A time-stamped trace was made by inserting calls to a trace routine at various points in the code. The trace information recorded included a time-stamp, a type, and possibly a copy of the RPC header part of the current message. (The trace was used for debugging as well as performance analysis.) Recording the trace information slowed down the RPCs, so the cost of tracing was measured and subtracted out of the results presented here.

The test performed during the detailed tracing was a series of Echo RPCs; the client sent 128 bytes to the server which then returned them. Two sets of trace results are given: initial results done before any system tuning was done, and another set that corresponds to the performance plotted in Figures 11 and 12.

The initial trace results are given in Table 2. The Average column is an average over 5 traces, including system overhead. The Best column estimates the time required for each step in the absence of external events. The estimates were made by looking at the raw trace records and picking the most common time for a task. Typically a task had the same time in most RPCs, but occasionally an external event would slow down the task. The scheduler, for example, wakes up once a second and recomputes priorities.

The total estimated overhead (from the Best column) was about 7400 microseconds. The Echo tests were also done with no trace records kept, and the average time for each Echo in that case was 8400 microseconds. Some of the remaining 1000 microseconds is spent in the network module when receiving the message. The rest is probably due to errors in using the Best column to estimate what happened in the absence of tracing.

These early results indicated that the process switch time and the time to output a packet were high. Switching from the dispatcher to the waiting process was 1100 to 1300 microseconds, and a complete process switch was taking about

Initial Trace of the Echo RPC (128 bytes echoed) Times in microseconds			
Where	Task	Average	Best
Client	Start		
	Allocate Channel	329	282
	Message Setup	392	286
	Request Output	1252	1148
	(Sub Total)	1973	1716
Server	Request Arrives		
	Server Chosen	128	128
	Server Notified	826	714
	Server Starts	2516	1275
	Reply Output	1606	1445
	(Sub Total)	5076	3562
Client	Reply Arrives		
	Client Notified	862	862
	Client Starts	1198	1145
	Client Returns	133	107
	(Sub Total)	2193	2114
	Total Overhead	9242	7392

Table 2. The early results from a time-stamped trace of an Echo RPC. The effect of taking the trace records themselves has been factored out of the data. The Average column is an average over 5 traces, while the Best column is an estimate of the best time for the task.

2200 microseconds! This was due to two expensive operations done during a process switch. The first is a weighted CPU usage computation that requires software multiplies and divides. The second is the setting up of an alarm for a time-slice; this requires reading a real-time counter, converting to seconds, and enqueuing a call-back routine. The time spent in the network module was also high. Some of this was due to the complex interface to the Intel controller, and some was due to excess error checking in the initial version of the code.

Both process switching and the network module were cleaned up in an effort to improve performance. The timer module was changed to work directly with chip intervals instead of seconds. The CPU usage computation was simplified. A new priority was added that eliminated the usage computation and the setting up of the time-slice during process switch, and server processes were set to run at this high priority. The network module was tuned by removing excess error checking, and by simplifying various data structures so they could be set up at system start-up. For example, the addresses passed to the Intel controller need to be byte swapped, and this can be done once because the network module re-uses the same

buffers.

The average of an untraced set of echoes improved from 8400 microseconds to 6050 usec after the these changes to the kernel. The results of the new detailed traces are given in Table 3. The best time to output a packet decreased about 520 usec on the server, and about 564 usec on the client. Another 915 usec is saved during the process switch to the server process because no CPU usage computation is done, and because no time-slice is set up. Also, allocation of the client channel sped up because it involved getting a monitor lock; the locking code has been re-written in assembly language.

However, the cost of handling the message on the client increased in spite of (or because of) the changes to the timer module and the CPU usage computation. The time on the client to set up the message is still high, and so is the client's context switch time. (This is the "Client Starts" entry in the table.) The client process runs at a normal priority so there is still a lot of scheduling overhead at process switch time. This poor performance reflects the state of the kernel's

Current Trace of the Echo RPC (128 bytes echoed) Times in microseconds				
Where	Task	Average	Best	Difference
Client	Start	(156)	(132)	(+52)
	Allocate Channel	293	236	-46
	Message Setup	473	350	+64
	Request Output	605	584	-564
	(Sub Total)	1371	1170	-546
Server	Request Arrives	(7293)	NA	
	Server Chosen	215	215	+87
	Server Notified	793	793	+79
	Server Starts	541	360	-915
	Reply Output	953	925	-520
	(Sub Total)	2502	2293	-1269
Client	Reply Arrives	(4972)	NA	
	Client Notified	1087	1087	+225
	Client Starts	1514	1410	+265
	Client Returns	176	165	+58
	(Sub Total)	2777	2662	+548
	Total Overhead	6650	6125	-1267

Table 3. Trace results after improving the server's process switch time and tuning the network module. The format of the table is the same as Table 2, except that the Difference column gives the difference between the Best column in Table 2 and Table 3.

development; most of the effort has been in the addition of new features, and much less effort has been spent analyzing and improving the performance. Micro-benchmarks that test various kernel operations need to be developed so that costly operations can be identified and tuned.

7. Comparisons to Other Systems

The Sprite RPC system incorporates two important features from the RPC work by Birrell and Nelson [BiN84] for the CEDAR/MESA system. The first is that it uses a special purpose network protocol that only uses 2 messages per RPC in the common case. The other feature is that the service procedure is executed at most one time during an RPC. This is because the reply message is acknowledged, either implicitly or explicitly. This allows Sprite services to implement locking and other operations that cannot be repeated. The main difference between the two systems is the way clients and servers are connected. In the CEDAR/MESA system the clients and servers are user processes that are bound together through the Grapevine [Bir82] name service. In the Sprite RPC system the binding is between a client channel and a server process. This binding is dynamic and is controlled by the server so it can allocate its server processes to active channels.

The Sun RPC [RPC85] system differs in that it uses existing protocols to transport its messages. It can use either TCP [TCP80], which is a reliable byte stream protocol, or UDP [UDP79], which is an unreliable datagram protocol. These may be less efficient than a special purpose protocol, but they can be used in inter-networks. Another important difference is that reply messages are not acknowledged. Instead, a server caches its recent replies and re-sends them when it detects a retransmission by the client. If the cache fills up, the oldest cached replies are discarded; this means that a service procedure could get executed more than once per RPC. Sun takes this approach because it prefers to implement *stateless* servers. A stateless service can have its service procedures executed more than once with no ill effect. Also, a stateless server can crash and re-boot and a client only experiences a delay; no state is lost. However, it also means that Sun's network filesystem [NFS85] does not implement file locking, for example.

Other systems have paradigms other than RPC for communication between hosts. The V kernel [Che84] is a good example of an efficient message passing kernel. Its performance is considered nearly optimal [LaZ84] so it provides a good standard with which to compare. Also, its use of messages is similar to the way the RPC protocol uses messages; the send operation is followed immediately by a receive operation that blocks the process until a reply arrives. These semantics were found to be useful and less error-prone than more arbitrary uses of messages.

Reliable byte stream protocols, TCP for example, are used by many systems for network communication. A byte stream protocol is built on top of an unreliable datagram protocol as is the Sprite RPC protocol. Its concern, however, is a reliable, uncongested, bi-directional flow of bytes between two points, while the

concern of the RPC protocol is the efficient exchange of the request and reply message. Note that it is possible to implement an RPC protocol on top of byte streams, but probably with less performance. The Sun RPC system can use TCP, for example.

8. Summary and Conclusions

Sprite uses remote procedure call as its basic form of network communication, and, for better performance, RPC is implemented on top of a special-purpose network protocol. Performance is emphasized so that remote services will be cheap enough to be used frequently. Three features of the RPC implementation are designed to increase its performance. (a) Implicit acknowledgment is used to reduce message overhead. (b) The buffering system is optimized for block transfer; a data block is only copied 3 times when going from a client process's buffer over the network to a server process's buffer. (c) Fragmentation, while it adds some complexity to the system, increases the performance of large data transfers.

Of these three techniques, implicit acknowledgment provides the best performance increase for RPCs that transfer a small amount of data. In this case the message overhead is high compared to the total time of the RPC so it is worthwhile to eliminate explicit acknowledgment messages. The other performance related features, fragmenting large transfers and saving a copy on large data blocks, significantly increase performance of large data transfers. With fragmenting the system achieves a bandwidth of 380 Kbytes/sec, while with no fragmenting the best bandwidth is only 135 Kbytes/sec. Eliminating one of the copies for large data blocks saves .8 msec/Kbyte, which is about one third of the marginal cost of sending each additional Kbyte (2.4 msec/Kbyte).

The Sprite RPC system differs from most other RPC systems in that it is used only for remote procedure call from one operating system kernel to another. A client process directs RPC requests to a server host, and it does not have to worry about setting up connections or addressing particular server processes; client channels and server processes are not visible outside the RPC system. Internally the RPC system sets up tempory bindings between client channels and server processes so that successive requests through a client channel implicitly acknowledge the server's replies. This arrangement also limits the amount of state information that the server side of the RPC system has to maintain. A server only has to keep track of channels that are activly using the server, and it does not have to know about all the client processes in the system.

Finally, there are some limitations to this RPC system. It is used by the Sprite kernel to execute service procedures in other Sprite kernels; there is no direct way to invoke user-level service procedures, although the RPC system could be used to implement that. Also, there is no stub compiler; the number of different service procedures is limited so it is not too painful to hand-code the required stubs.

Appendix A
Message Control Information

This appendix contains the format of the control information included in each message, and a short explanation of each field. The control information comes after the header used by the low level transport protocol. For the basic Ethernet protocol, for example, there is a 14 byte header: 6 bytes of destination address, 6 bytes of source address, and a 2 byte protocol identifier.

Per Message Control Information	
Field	Width
Flags	16 bits
Client Host ID	32 bits
Server Host ID	32 bits
Channel ID	16 bits
Server Process ID	16 bits
RPC Sequence Number	32 bits
Number of Fragments	16 bits
Fragment Mask	16 bits
RPC Command/Error	16 bits
Boot ID	32 bits
Data 1 Size	16 bits
Data 2 Size	16 bits
Data 1 Offset	16 bits
Data 2 Offset	16 bits
Total Size	36 bytes

Table A1. The layout of the control information that is at the beginning of every message sent by the RPC protocol. The type of the message is encoded in the Flags field, and for simplicity all fields are present in all types of messages.

Flags - 16 bits

The Flags field includes the message type and some flags which may or may not be present. The message types are REQUEST, REPLY, PARTIAL_ACK, and EXPLICIT_ACK. The flags are SERVER (the message is for a server process), ACK_REQUESTED (return an explicit acknowledgment message to the sender), ERROR (the service procedure returned an error), and LAST_FRAGMENT (this is the last fragment of the message).

Host IDs - 32 bits each

The client and server Sprite host IDs uniquely identify the client and server hosts. Currently Sprite IDs are assigned locally, but in the future they may follow the Internet addressing standards [IP80].

Channel ID - 16 bits

Identifies a particular client channel on the client host.

Server Process ID - 16 bits

In messages from the server, this is the ID of the server process that is handling the request. In messages from the client, this is the latest server process ID received from the server; it is used as a hint by the dispatcher on the server to quickly match incoming messages to the correct server process.

RPC Sequence Number - 32 bits

The sequence number is the same for all messages pertaining to the same RPC.

Number of Fragments - 16 bits

This is non-zero if the message is fragmented; it indicates how many fragments make up the message.

"Flags" Field Encoding		
Name	Bits	Comment
ACK_REQUESTED	0x0001	Acknowledgment of message requested
LAST_FRAGMENT	0x0002	This is the last fragment
ERROR	0x0008	Error return from the service procedure, the command field contains the error code
SERVER	0x0010	Packet is for a server process
REQUEST	0x0100	Normal command request
PARTIAL_ACK	0x0200	A partial acknowledgement
EXPLICIT_ACK	0x0400	An explicit acknowledgement
REPLY	0x0800	A reply from a server

Table A2. The encoding of the Flags control field.

Fragment Mask - 16 bits

The I'th fragment of a REQUEST or REPLY message is identified by setting the I'th bit of this field. Messages are limited to 16 fragments. This field is zero if the message is not fragmented. If the message type is PARTIAL_ACK, this field contains the summary bitmask that indicates which fragments the receiver has.

RPC Command/Error - 16 bits

This field contains the ID of the remote service procedure to execute. In REPLY messages this field contains an error code if the ERROR flag is set.

Boot ID - 32 bits

This is a time-stamp chosen by the client host when it re-boots. When this changes from one RPC to the next the server knows the client has re-booted.

Data Sizes - 16 bits each

These fields indicate how much data is in the two data parts of the message.

Data Offsets - 16 bits each

These fields indicate the relative position of the data areas in the complete message. For non-fragmented messages these fields are always zero.

Appendix B
RPC Sequence Numbers and Client Crashes

A client kernel initializes its sequence number at boot time (to 1) and then increments it for each new RPC by a client process. The servers in the system need to be informed somehow that the client has rebooted and reset its sequence number; otherwise they might discard the client's messages because of their low sequence numbers. Instead of using a reliable broadcast to reach all the servers in the system, or doing a special RPC with each one, the protocol uses a time stamp, called the *boot ID*, to detect when a client has rebooted. The client kernel initializes its boot ID to the real-time clock at boot time, and it doesn't change until the next reboot. When a server sees a new boot ID from a client, it resets its notion of the client's current sequence number.

Client's with no real-time clock need some other way to initialize their boot ID to value that is distinct from what they chose the last time. The implementation allows a boot ID of zero; if it is zero, the server accepts the request message regardless of the sequence number. This trick is used to do an RPC that returns the time so the client can initialize its boot ID.

Appendix C
Partial Re-send on Fragment Loss

A system of partial re-send and partial acknowledgment is part of the Sprite RPC network protocol. The current implementation is not perfect, however, so it is presented in this Appendix with some suggestions for future improvements.

When doing partial re-send there are series of related problems. The first is how to tell the sender of a fragmented message which fragments have been successfully received. This is done by returning a partial acknowledgment message that contains the receiver's summary bitmask. The summary bitmask has a bit set for each fragment that has been successfully received.

The next problem is when should the receiver return a partial acknowledgment. In the current implementation, the first time a receiver checks for missing fragments is when the last fragment of a message arrives. The last fragment is flagged with the LAST_FRAGMENT flag so the dispatcher knows when to check. The flag is required because on a partial re-send the last fragment sent may not be the last fragment in the message. If the dispatcher detects missing fragments at this point it returns a partial acknowledgment to the sender.

If the last fragment is lost, the situation is handled differently on the client and the server. The client process will timeout waiting for its reply, and at that point the it checks to see if it is receiving a fragmented reply. If it is, it notes the number of fragments it has received and waits again. If it times out again and has not received more fragments then it decides that some are missing and it returns a partial acknowledgment to the server.

The servers, however, are passive, and they do not set up timeout periods. Instead, a server has to wait for fragments to arrive and decide when some fragments have been lost. It does not detect the loss of the trailing fragment until the client retransmits its request and duplicate fragments begin to arrive. Each time a duplicate fragment arrives the server returns a partial acknowledgment. Because this can result in several partial acknowledgments, the client does not resend right away when it gets a partial acknowledgment. Instead, the client dispatcher saves the summary bitmask in the state of the client channel, and the client process waits until its timeout period elapses before resending. This is done so that it resends with the latest summary bitmask from the server.

The loss of the last fragment by the server can result in very poor performance if it happens very often. The situation can be improved by setting up a timer on the server when a fragmented message begins to arrive. The current implementation does not do this because it is extra overhead to enqueue call-back routines and then dequeue them after all the fragments arrive safely. Not having

a timer on the server, however, creates such a poor worst case that it is probably worth it for the server to set up a timer when receiving a fragmented request.

The other possible modification to the current implementation would be to not have the client resend a fragmented request message upon timeout. Instead, it could send a probe message to which the server could reply with a partial acknowledgment. Also, when re-sending a fragmented message, both the client and the server could introduce delays between fragments so the receiver has a better chance of getting all the fragments.

Appendix D
Example Stub Procedures

The client stub for the remote Read() procedure is given in Figure D1. The stub uses a local buffer, *readParams*, to hold the three input parameters to the procedure. It sets up the first data area of the request message to reference this local buffer, and the second data area is empty. By convention, the second data area is used for large data blocks. The client stub also sets up buffer pointers for the two data areas of the reply message. In this case both data areas are used. The first contains the return parameter, *amountRead*, and the second contains the bytes read from the file. Because there are only two parameters, no local buffer is needed by the stub. The dispatcher will copy the *amountRead* value and the data block directly to the client process's buffers for those values.

Figure D2 shows the corresponding server stub for the remote Read procedure. The dispatcher copies the three parameters in the first data area into a buffer previously set up by the server process. The server stub references this via *requestMessagePtr->data1*. The stub then calls the LocalRead() procedure which sets the *buffer* variable to point to a buffer in the filesystem's cache. Finally, the server stub sets up pointers to the two buffers containing the data for the two data areas of the reply message. In this case, the two buffers are the one in the filesystem cache, and the local variable *amountRead*. After the server stub returns, the server process sends the reply message to the client process.

```
RemoteRead(filePtr, offset, size, buffer, amountReadPtr)
    FsFile *filePtr;
    int offset;
    int size;
    char *buffer;
    int *amountReadPtr;
{
    struct ReadParams {
        int fileID;
        int offset;
        int size;
    } readParams;
    RpcMessage requestMessage;
    RpcMessage replyMessage;
    int returnStatus;

    readParams.fileID = filePtr->fileID;
    readParams.offset = offset;
    readParams.size = size;

    requestMessage.command = RPC_READ;

    requestMessage.data1 = (char *)&readParams;
    requestMessage.size1 = sizeof(readParams);
    requestMessage.data2 = (char *)0;
    requestMessage.size2 = 0;

    replyMessage.data1 = (char *)amountReadPtr;
    replyMessage.size1 = sizeof(int);
    replyMessage.data2 = buffer;
    replyMessage.size2 = size;

    returnStatus = RPC_Call(filePtr->serverID, &requestMessage, &replyMessage);

    return(returnStatus);
}
```

Figure D1. The client stub procedure for the read service procedure. Routines in the Sprite kernel call a general Read() procedure, and Read() calls RemoteRead() if the file is remote.

```
SrvRead(requestMessagePtr, replyMessagePtr)
    RpcMessage *requestMessagePtr;
    RpcMessage *replyMessagePtr;
{
    struct ReadParams {
        int fileID;
        int offset;
        int size;
    } *readParamsPtr;

    char *buffer;
    int amountRead;
    int returnStatus;

    readParamPtr = (struct ReadParams *)requestMessagePtr->data1;

    returnStatus = LocalRead(readParamPtr->fileID, readParamPtr->offset,
                             readParamPtr->size, &buffer, &amountRead);

    replyMessagePtr->data1 = (char *)&amountRead;
    replyMessagePtr->size1 = sizeof(int);
    replyMessagePtr->data2 = buffer;
    replyMessagePtr->size2 = amountRead;

    return(returnStatus);
}
```

Figure D2. The server stub procedure for the read service procedure, LocalRead(). LocalRead() is also called by the general Read() procedure when it reads from a local file.

References

[Bir82] A. D. Birrell, "Grapevine: An Exercise in Distributed Computing", *Comm. of the ACM 25*, 4 (Apr. 1982), 260-274..

[BiN84] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39-59.

[ChZ83] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", *Proceedings of the 9th Symp. on Operating System Prin., Operating Systems Review 17*, 5 (Nov. 1983), 129-140.

[Che84] D. R. Cheriton, "The V Kernel: A software base for distributed systems.", *IEEE Software 1*, 2 (Apr. 1984), 19-42.

[UDP79] "DoD Standard User Datagram Protocol", *Internet Working Group IEN 8*, May 1979.

[TCP80] "DoD Standard Transmission Control Protocol", *Internet Working Group IEN 129*, Jan 1980.

[IP80] "DoD Standard Internet Protocol", *Internet Working Group IEN 128*, Jan 1980.

[LaZ84] E. D. Lazowska and J. Zahorjan, "The Performance of Diskless workstations", Technical Report 84-06-01, University of Washington, June 1984.

[McK84] M. K. McKusick, "A Fast File System for UNIX", *ACM Transactions on Computer Systems 2*, 3 (Aug. 1984), 181-197..

[MeB76] R. M. Metcalfe and D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM 19*, 7 (July 1976), 395-404.

[NFS85] *Sun's Network File System*, Sun Microsystems, 1985.

[RPC85] *RPC Protocol Specification*, Sun Microsystems, 1985.

[WeO85] B. B. Welch and J. K. Ousterhout, "Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem", Technical Report UCB/Computer Science Dpt. 86/261, University of California, Berkeley, Oct. 1985.