An Implementation of a Remote Procedure Call Protocol in the Berkeley UNIX* Kernel

Karen White

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

Abstract

This report describes the design and implementation of a request/response protocol for the DoD Internet protocol hierarchy. The protocol enforces request/response semantics and can be used as part of a remote procedure call package. The protocol is not limited to this use, but most design decisions were made with this application in mind. The protocol was implemented in the Berkeley UNIX kernel and integrated into the inter-process communication mechanism. Measurements indicate the protocol is two to three times faster than similar protocols implemented in user level code. The protocol is as fast as other kernel protocols; users will therefore be free to choose the protocol which best matches the semantics of their distributed applications.

*UNIX is a trademark of Bell Laboratories

This work was sponsored the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

ACKNOWLEDGEMENTS

Special thanks are owed to Eric Cooper for his continual guidance and support of this work. As the head of this research project, his constant help and motivation were invaluable.

I would also like to thank Domenico Ferrari, my research advisor, and the members of the PROGRES research group.

Finally, I would like to thank my family for their ongoing encouragement.

1. Introduction

Remote procedure call (RPC) provides a simple method for a user to access routines that reside on different machines. Nelson defines RPC as follows:

> Remote procedure call is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel [14].

The semantic similarity of RPC to normal procedure call enables a user to use inter-process communication capabilities while preserving the model that the user already has of his program, namely procedure calls.

This paper describes the design and implementation of a request/response protocol that is to be used as part of a remote procedure call package. The protocol was implemented in the Berkeley UNIX kernel and integrated into the inter-process communication mechanism. Observations are made about this communication mechanism and the Berkeley UNIX socket interface. Also included is a summary of measurements of the speed of the protocol.

The remainder of this section presents background material on RPC and the motivation for the present work. The rest of the paper contains an overview and functional description of our RPC protocol, some implementation details and decisions, observations about the UNIX socket mechanism, measurements of our protocol, and discussions of the application of RPC to distributed systems.

1.1. Background

RPC was developed to facilitate the design of distributed programs by hiding the details of the underlying network from the user. RPC allows a programmer to concentrate on designing programs without worrying about where the pieces of the program may be distributed. Thus, a remote procedure call looks like a local procedure call. Stub procedures, generated at compile-time from an interface specification, hide all the details of communication.

To invoke a remote procedure, the client stub on the calling machine packages the procedure name and parameters into a message and sends the message to the remote machine. When the server stub on the remote machine receives a request message, it unpacks the message, invokes the named procedure, packages the result into a message, and sends the result back to the caller. The caller then returns the result to the user program.

The work described in this paper is an extension of Cooper's work in fault-tolerant distributed programming [4,5,6]. We implemented his RPC protocol in the Berkeley UNIX kernel. Other work in the area includes Nelson's Ph.D. work [14], Xerox's Courier [21], a remote procedure call standard, and Birrell and Nelson's RPC facility at Xerox PARC [3]. In addition, Sun Microsystems has an RPC facility that handles broadcasting [19].

1.2. Motivation

In addition to being simple to use, a mechanism for constructing distributed programs must be relatively fast if it is expected to be used extensively.

Larus [11] made performance studies of two versions of Courier that had been modified for 4.1BSD. Cooper's version used TCP, a byte-stream protocol, for its transport protocol. Another version, PCourier, was developed by Larus using UDP, a datagram protocol, for its transport layer. Larus found PCourier to be three to four times faster than the TCP-based Courier (when discounting PCourier's excessive time-out delays). He attributed this difference in performance to the use of datagrams as opposed to byte streams for the underlying protocol. Although PCourier demonstrated favorable results, it was not considered as a possible replacement for Courier because it was based on unreliable datagrams. Unreliable datagrams are unsuited for a production system because their delivery is not guaranteed and they may arrive out of order. Thus, Larus predicted that a reliable packet protocol would speed Courier up, though not as fast as his UDP implementation.

Cooper's Circus system implements a UDP-based reliable packet protocol in user code [5]. Kupfer compared the performance of a Circus-based remote instrumentation program with a TCP-based version of the same program [10]. He concluded there was too much overhead involved when using the Circus-based version for it to be usable in a production system. He also predicted that an implementation of Circus's communication protocol in the Berkeley UNIX kernel would speed it up considerably.

Although a specialized protocol is not necessary for an RPC implementation, it is necessary for a *user acceptable* implementation. After reading the findings of Larus and Kupfer, we hoped to achieve performance gains for Circus by implementing Cooper's paired message protocol in the Berkeley UNIX kernel.

2. The Protocol

One approach to implementing an RPC protocol is to build it on top of a reliable message protocol. A reliable message protocol guarantees the sending of a single variable-length message in one direction. Assuming that unreliable datagrams are used, the reliable message protocol would have to acknowledge each message to ensure reliability. (An acknowledgement indicates that the last sent message has been successfully received.) Thus, one remote procedure call using reliable messages would require the call and its acknowledgement and the return and its acknowledgement, or a total of four messages.

An alternative to this approach is to implement a request/response protocol. Since the protocol enforces a request/response pairing of messages, the response can serve as an acknowledgement of the request. Furthermore, the next request can be used to acknowledge the previous response. This reduces the number of messages exchanged. A sequence of calls made between the same client and server will require two messages per call, whereas using a reliable message protocol as described above would require four messages per call. This is the approach taken by previous implementors of RPC protocols and it is the one followed here [3,5,14].

The rest of this section contains a general overview of our protocol followed by a more detailed functional specification. The following definitions will facilitate the reading of the protocol description.

A *segment* is a block of data attached to a header. It is the unit of information exchanged between RPC protocol implementations on different communicating machines. A segment has a fixed maximum size. It is possible for segments to contain no data, just control information.

A *message* is a variable length sequence of bytes that corresponds to a request or a response. A message consists of one or more segments, depending on the amount of data sent. Messages are sent or received by the user (either the client or server program).

A *probe* is a dataless segment sent by the RPC protocol to determine if there has been a break in communication between the client and server.

An *acknowledgement* (ACK) is a dataless segment sent by the RPC protocol to acknowledge the successful reception of a segment or a probe.

For the purposes of this paper, the phrases *request/response* and *call/return* have the same meaning and they will be used interchangeably. *Client* will be used to refer to the sender of the request and *server* will be used to refer to the sender of the response. The terms *segment* and *packet* will also be used interchangeably.

2.1. Overview

The protocol is an implementation of Cooper's Circus protocol [5] modified for the Berkeley UNIX kernel [8]. Cooper's protocol closely follows the RPC protocol of Birrell and Nelson [3]. It is a reliable paired message protocol: it enforces a request/response pairing of variable-length messages and guarantees their reliable delivery. The protocol is connectionless in the sense that

no initial handshaking is done to initiate a connection and there is no terminating exchange of messages to delete a connection. Little state is maintained between call/return pairs. The protocol is asymmetrical. The asymmetry is most obvious when handling probing. Probing is the sending of packets between client and server to detect whether a break in communication has occurred. The asymmetry will be pointed out in the protocol description.

Our protocol is implemented on top of UDP [15] in the DoD Internet protocol hierarchy, as illustrated in Figure 1.

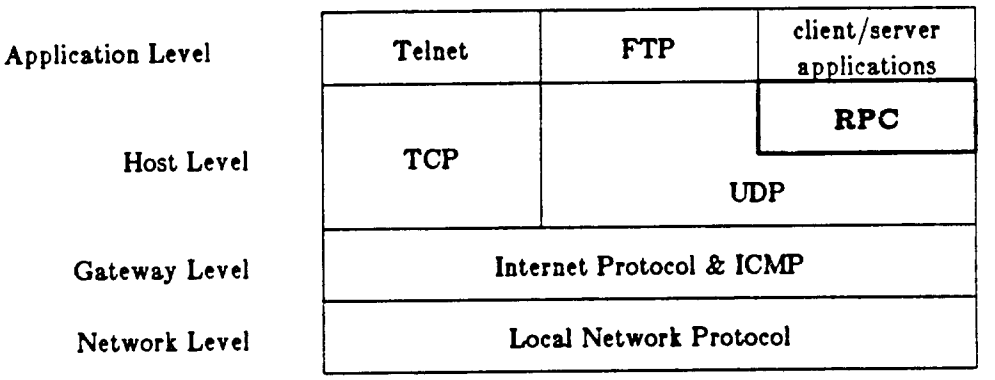

Figure 1: Protocol Hierarchy

Message transmission by the RPC protocol consists of receiving data from the user in one or more segments, prepending an RPC header to the segment, and handing the segment to UDP to be sent as a datagram. When UDP receives an RPC segment, it hands it up to the RPC input routine which strips off the header and delivers the data to the user.

There are two aspects of flow control in our protocol. There is local flow control between the user and the kernel, and there is remote flow control between two kernels running on communicating machines. Local flow control is implemented by suspending (blocking) a user process if the last sent segment has not been acknowledged by its peer. When receiving a message, the user process is blocked until data is available to deliver to the user. Remote flow control is implemented by using ACKs. When a segment has been successfully acknowledged, a kernel knows it can send the next segment. Similarly, once received data has been delivered to the user, the kernel knows it is ready to receive more data.

Reliability is guaranteed by using timers, sequence numbers, and ACKs. Each call/return pair of messages has a unique call number, and each segment within a message has a unique segment number. The call and segment number of the most recently sent or received segment are stored as state information during calls. Between calls only the last call number is stored. Duplicated, lost, or dropped segments are handled by retransmissions and explicit ACKs when necessary. Duplicate packets are determined by comparing the call and segment numbers in the packet header with that stored as state information. If the numbers are the same, or if the numbers in the incoming packet are smaller (using an unsigned 32 bit comparison), the message is a duplicate and is dropped. If an acknowledgement of a sent packet is not received within a predetermined amount of time, a retransmission timer expires and the last sent packet will be retransmitted with a request for an explicit acknowledgement. Thus, lost or dropped segments will be resent. A segment is dropped if the previously received segment has not been delivered to the user because we allocate only a single-segment buffer per connection in the kernel. This means that only one segment can be stored in the kernel at any given time.

Damaged segment headers or data are detected by the checksumming implemented at the UDP level.

2.2. Functional Description

An RPC header is prepended to every segment before it is handed down to the UDP output routine. The RPC header is illustrated in Figure 2.

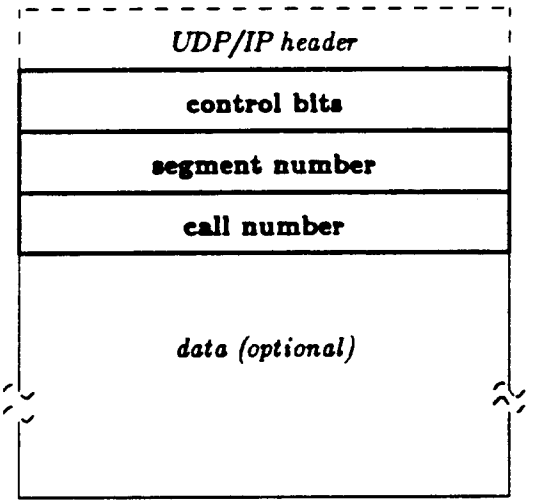

Figure 2: RPC header

The control field is used to distinguish between the different types of segments that may be sent. The field also tells the receiver if the segment should be explicitly acknowledged. The following variables are represented as consecutive bits in the control field. They are ordered from least significant to most significant bit.

PLEASE_ACK	An explicit ACK is to be returned if this bit is on.
ACK	If set, the segment is an explicit ACK. ACKs are never piggybacked with data. (An implicit ACK, described in section 2.5, does not have the ACK control bit set.)
LAST_SEGMENT	If set, the segment is the last segment of a multiple segment message or the only segment of a single segment message.
PROBE	If this bit is set and no others are set, the segment is a probe. If this bit is set in conjunction with the ACK bit, the segment is an ACK of a probe.

Several combinations of these bits are possible. If the LAST_SEGMENT bit is the only bit set, the segment is either the only segment of a single segment message, or the last segment of a multiple segment message. An explicit acknowledgement is not requested because a quick response segment will serve as the acknowledgement of this segment. If the PLEASE_ACK bit is the only bit set, the segment is any but the last segment of a multiple segment message. It could also be a retransmission of any but the last segment of a multiple segment message. If the LAST_SEGMENT and PLEASE_ACK bits are set, the segment is a retransmission of the last segment of a message. This segment could be sent if the response message was not sent quickly enough to serve as an implicit acknowledgment of the sent message. If the ACK bit is the only bit set, the segment is an acknowledgement of the last received segment. If both the ACK and PROBE bits are set, the segment is an acknowledgement of a probe. If the PROBE bit is the only bit set, the segment is a probe.

The segment number field contains the segment number of the segment being sent or the segment being acknowledged. For probing, this field is ignored. Segments are numbered starting with 1. Thus, a single segment message will have its only segment numbered 1.

The call number field contains the unique call number that distinguishes call/return pairs. The call number must be a monotonically increasing number to insure the detection of duplicate or delayed messages.

2.3. Sending a Segment

Segment transmission is treated in the same way by the client and server. It is best defined by describing the treatment of five different segment types: a segment of a single segment message, a segment of a multiple segment message, a retransmission, an ACK, and a probe. The current call number and segment number will be placed in the RPC header of all segments unless stated otherwise. A new call number is obtained for each call/return pair of messages. As previously mentioned, message segments are numbered starting with the number 1. After properly filling in the header, the segments are handed down to the UDP output routine.

When sending data, the data is stored in a send buffer until an acknowledgement of the successful reception of this data by the peer has been obtained. The user process is blocked (suspended) if it tries to send another segment before the last sent segment has been acknowledged.

Sending the segment of a single segment message requires setting the LAST_SEGMENT control bit.

Sending a segment of a multiple segment message is handled like the segment of a single segment message if it is the last segment to be sent. For all but the last segment of the message sent, the PLEASE_ACK control bit must be set in the header.

When retransmitting a segment, a copy of the last segment sent is resent with the PLEASE_ACK control bit set. The call number and sequence number are unchanged.

Sending an ACK merely involves setting the call number and segment number of the header to the value of the last received segment and setting the control bit to ACK. No data is sent with the header. If a probe is being acknowledged, the segment number field is ignored and both the ACK and PROBE bits must be set in the control field.

Sending a probe involves setting the call number to the current call number and setting the PROBE bit in the control field. The segment number field is not used.

2.4. Receiving a Segment

The reception of a segment is also treated in the same way by the client and by the server. It is best explained by looking at the following cases. When a data segment is received by the RPC protocol, state information, including call and segment numbers, is updated and the data is stored in a receive buffer. The data remains in this buffer until it is delivered to the user. Upon data delivery, the receive buffer is cleared. This occurs when the user requests the data with some type of receive system call. If another data segment arrives before the previous data has been delivered to the user, the new segment is dropped. If the PLEASE_ACK bit was set in the new data segment header, the previously received data segment is acknowledged instead.

If a data segment is received with the PLEASE_ACK control bit set, an acknowledgement is sent of the last received message. If the segment number is one, the segment is the beginning of a new message. If the segment is the first segment of a message, and if the last segment sent to the peer has not already been acknowledged, then the received segment is treated as an implicit ACK of the last sent segment. Implicit ACKs are described in more detail in Section 2.5.

If a retransmitted segment is received, it will either appear to be the next correct segment (if the previous segment was lost), or it will appear to be a duplicate (if its call and segment number are the same as the previously received data segment's call and segment number). If it is the former, it is treated as a normal data segment. If it is a duplicate, the segment is dropped and an ACK is sent as requested. (A retransmitted segment always has the PLEASE_ACK control bit

set.) Duplicates are detected by comparing the call and segment number fields in the segment header to those in the stored state information.

When an ACK is received, its call and segment numbers are compared to those of the previously sent data segment. If they are the same, and if the previously sent segment has not already been acknowledged, then the segment is marked as acknowledged, and the data is removed from the send buffer. Recall that the data was stored in the send buffer when the segment was sent. In addition, any processes that were suspended until the send buffer became empty are woken up.

When a PROBE ACK is received, its call number is compared to that stored in the current state information. If it is the same, the probe counter is properly updated (to be defined in Section 2.6). If the call numbers are not the same, the PROBE ACK is dropped.

If a probe is received, and the call number is the same as the one stored in the current state information, a PROBE ACK is sent to the peer to indicate the receiver is still alive. If the call numbers do not match, the probe is dropped.

2.5. Acknowledgements

Acknowledgements are used to indicate the successful reception of a data segment or probe. This section will only address acknowledgements of data segments. A segment may be acknowledged in two ways. It may be acknowledged either with an explicit ACK segment, or with an implicit ACK. As mentioned earlier, a response is treated as an implicit ACK of a request and a subsequent request is treated as an implicit ACK of a response. This is assuming that the requests and responses are received before the previous segment was retransmitted with a PLEASE_ACK request.

The reception of an implicit ACK and an explicit ACK are handled in the same way. When receiving an ACK, if the previous segment has not been acknowledged, the segment is marked as acknowledged and the data that was stored in the send buffer is released. In addition, any processes that were suspended until the send buffer was cleared are woken up.

Each segment except the last of a multiple segment message must be explicitly acknowledged. This is done so that the kernel only needs to keep one buffer of unacknowledged sent data. Thus, there is at most one unacknowledged segment at any point in time. The last segment of a multiple segment message is treated the same as a single segment message. An explicit acknowledgement is not requested the first time the message is sent. If a response segment is not received within a predetermined amount of time, the segment is resent requesting an explicit acknowledgement. If a response segment is received, it serves as an implicit ACK and is treated as described above. In the expected case of quick remote procedures, this implicit ACK optimization eliminates the need for explicit ACKs and cuts the number of packets in half.

2.6. Probing

Probing is used to detect failures in communication between the client and server. Failures may result from network partitioning, a machine crash, or process termination. A client probes the server if there is a delay between the client's request and the server's response. This is necessary because once the client's request has been acknowledged (after a retransmission explicitly requesting an acknowledgement), the client would otherwise not receive any communication from the server before the response is returned. Thus, the client would not be able to distinguish between communication failures and slow server responses. This is in keeping with the traditional semantics of a procedure call - the caller waits indefinitely for the return. The server, on the other hand, does not probe the client between calls because the client may no longer require the services of the server. Thus the protocol is asymmetrical with respect to probing.

Both the client and the server must probe when receiving multiple segment messages to distinguish between failures and slow responses from the user processes. Probing is started after one segment has been received and continues until the next segment is received. After receiving the last segment of the message, probing is no longer needed.

We followed Birrell and Nelson's probing strategy by providing exponential backoff when sending probes. This continues until reaching five minutes, at which point probes will be sent every five minutes. More details of our implementation can be found in the next section.

2.7. Timers

This section presents details of how timers are used in our protocol implementation to handle breaks in communication and lost, dropped, or duplicated packets.

All of our timers are implemented by counters. The value of a counter corresponds to the number of 500 millisecond interrupts that have occurred since the timer was started. When a timer reaches a certain value (or expires), the actions described below occur.

A retransmission timer is needed to identify when to retransmit a message. This timer is set after sending a segment and cleared upon receiving an acknowledgement for the sent packet. A counter keeping track of the number of times the same packet has been retransmitted is also necessary to help determine when a possible failure in communication has occurred. It is referred to as the retry counter. After sending a packet, if an implicit or explicit ACK has not been received when the retransmission timer expires, we retransmit the packet with a PLEASE_ACK request and increment the retry counter. We will retransmit up to MAX_RETRY times before assuming a failure in communication has occurred. After MAX_RETRY retransmissions have occurred with no ACK, an error is reported to the user.

A timer is also needed to notify us when to probe. The timer is started after one segment of a multiple segment message has been received and continues until the next segment is received. After receiving the last segment of the message, probing is no longer needed. The timer is also started when a client is waiting for a response from a server. After the reception of the client's request has been acknowledged by the server, the client probes the server until receiving a response. If a probe is not acknowledged, the probe timer is reset to its initial value and the probe is retransmitted up to MAX_RETRY times before assuming a break in communication has occurred. When a probe is acknowledged, the probe counter is doubled until it has a value of 5 minutes. Exponential backoff of probes was implemented to reduce the number of interruptions made to the server's kernel.

Finally, we have a no activity timer that is set when a client or server is between calls or after communication has been lost. The no activity timer represents the amount of time state information will be maintained after the last known activity on the socket has transpired. The socket's state information must be maintained for the duration of the timer to ensure lost, delayed, or duplicated packets will be handled correctly. It is also needed to ensure the proper treatment of a normal client closing its socket. If the state information disappeared as soon as a client received its last response, the server would retransmit the response with a PLEASE_ACK request and never get an acknowledgement. This would cause the server to think the client never got the response and thus return an error when actually the client had received the response the first time it was sent.

2.8. Errors

The user is informed of suspected breaks in communication with error messages returned from the socket system calls. An error messsage indicating that the remote machine may be down is returned to the user if probing times out or if a message is retransmitted the maximum number of times without success. Note that errors are passed up from the kernel to the user on the same machine. No error messages are sent from one kernel to another. Once a break in communication has occurred, both sides will eventually be notified when one of the several timers

time out. If a user does not follow the semantics of the RPC protocol (if the user attempts to send or receive data in the wrong order) an error message will also be returned. Note that the socket type is used by the local kernel to enforce the protocol semantics.

2.9. Examples

The protocol is best illustrated by giving an example scenario of the packets exchanged for a single segment message and a for a multiple segment message. These are similar to Birrell's and Nelson's examples [3]. In our examples, packets are numbered in the order that they are sent. Time is depicted as moving forward when reading from the top to the bottom of the picture. The capitalized terms indicate which control variables are set in the segment headers when the segments are sent. The comments in parentheses indicate what else is going on besides packet exchanges.

Figure 3 demonstrates the exchange of messages for a single segment call followed immediately by another with no time out delays. The client sends a call request to the server, and the server sends the result back to the client after invoking the procedure. The return is treated as an implicit ACK of the call. The client then immediately makes another call to the same server, so the second call is treated as an implicit ACK of the first call. The second call then proceeds as the first. This example demonstrates the minimum number of messages exchanged.

Figure 4 demonstrates the exchange of messages for a multiple segment call. This figure tries to depict each type of time out that can cause extra message transmissions and exchanges. The first call segment is sent by the client but never received by the server. After a retransmission time out, the first call segment is sent again. The server explicitly acknowledges this segment by sending an ACK, which is received by the client. The client then sends the last segment of the call, not requesting an acknowledgement. This segment is lost, so the client times out again and retransmits, this time requesting an explicit acknowledgement. The server receives this message and acknowledges it successfully. The server takes a long time to execute the procedure, so the client must start probing. After probing, the server responds with the results of the call. The

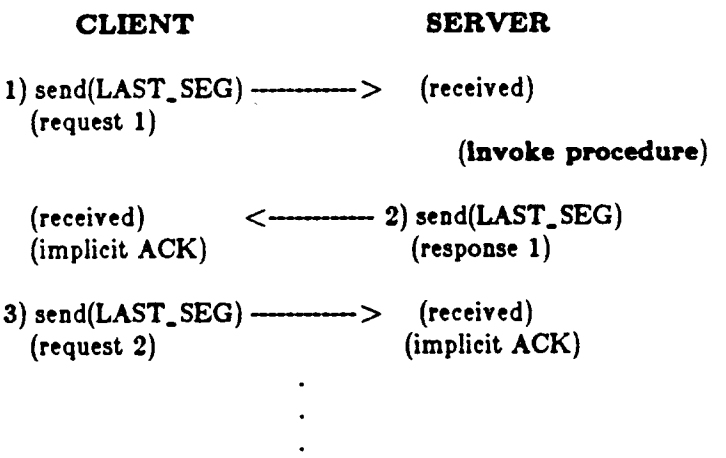

Figure 3. Single Segment Call and Return

client receives the response, closes the socket, and starts the no activity timer. The server must retransmit the response since there was no subsequent call made by the client that would act as an implicit ACK. When the server retransmits the result, it is explicitly acknowledged by the client because state information has been maintained for the duration of the no activity timer.

We have not shown the client or server probing when they are receiving multiple segment messages. This is handled in the same manner as probing between the call and return.

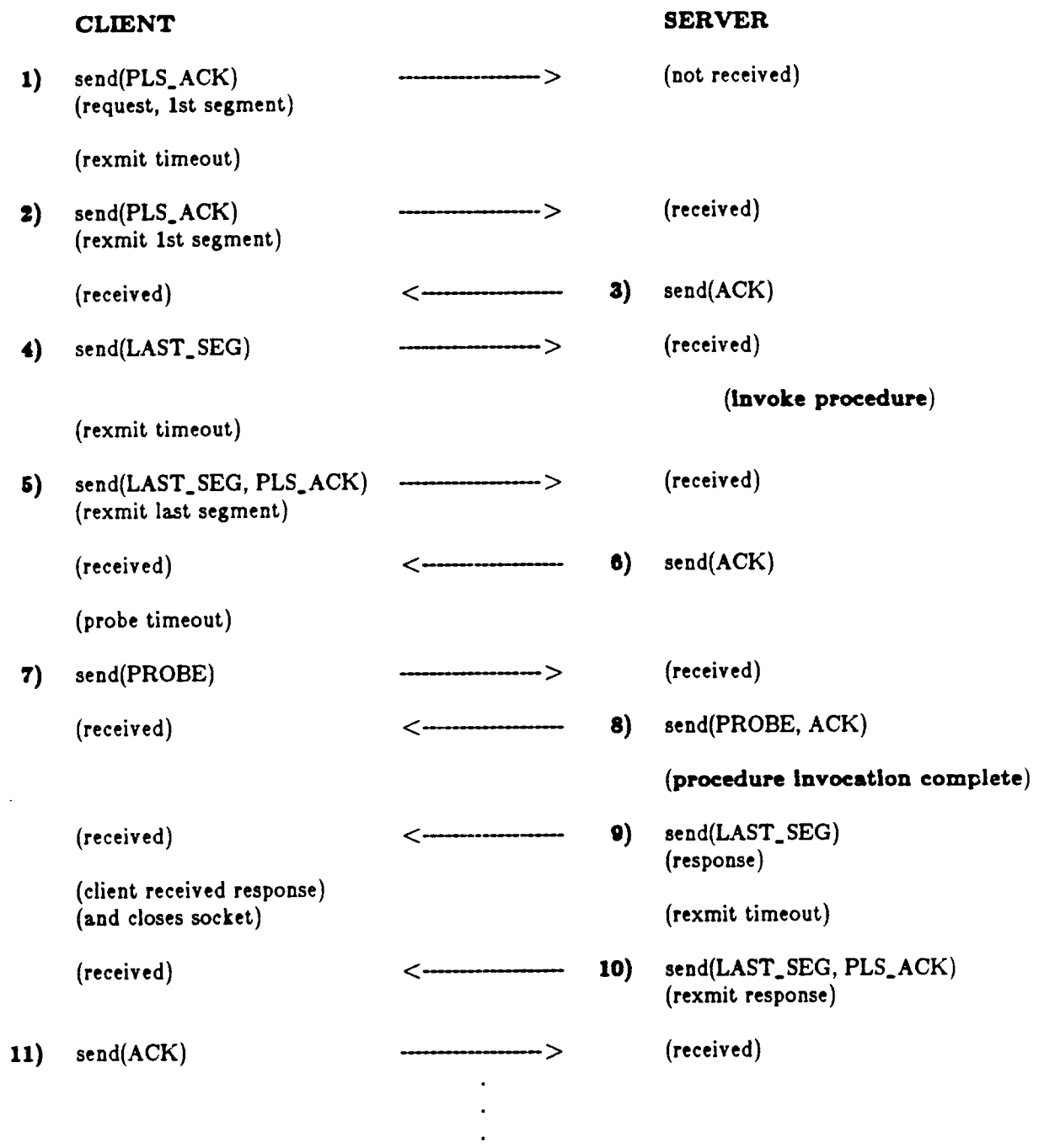

Figure 4: Multiple Segment Call, Single Segment Return
with Probing and Retransmissions

3. Design Decisions and Implementation Details

The first paragraph of this section gives a brief synopsis of the decisions that will be addressed in the later parts of this section. It is followed by a simplified view of how sockets are implemented in Berkeley UNIX.

Our RPC protocol was implemented on top of the DoD User Datagram Protocol (UDP) [15]. It uses a nonblocking send for the first segment sent and blocks on subsequent sends and receives until the previously sent segment is acknowledged. (A blocking send or receive suspends the user process if the operation cannot be performed, whereas a nonblocking send or receive returns immediately.) Local flow control (between user and kernel) is implicit in single segment messages and forced in multiple segment messages by use of the blocking sends and receives. Checksumming is handled by UDP. The client is required to call the *connect* system call before sending data, while the server is connected implicitly to the client upon receipt of the first message. The user is responsible for segmenting messages, for sending the segments, and for gathering the segmented messages on the receive side. (This is typically handled by the stub procedures linked into the user's programs.)

The following is a simplified view of Berkeley UNIX sockets. For details of the interprocess communication (IPC) mechanisms see [12,13]. A socket is an endpoint of communication. For two processes to communicate, each creates a socket of the same socket type, and messages are transferred between the two sockets. The message may either contain the address of where it is going, or two sockets may have been connected prior to exchanging any messages. If two sockets are connected, no address is needed when exchanging messages. Data may be read to and written from sockets. This is similar to the way data is read to and written from files. Rules governing the exchange of data are determined by the protocol and socket type. The socket type defines the semantics of communication, while the protocol enforces it. The protocol and socket type are assigned at socket creation time.

Each transport-level protocol (such as UDP or TCP) usually maintains its own queue of internet protocol control blocks (INPCB). An INPCB contains local and foreign address information as well as a pointer to a protocol-specific protocol control block (PCB) and a pointer to a socket data structure. The PCB contains protocol-specific state information. The socket data structure contains socket-specific data such as the socket type, a socket error reporting field, and receive and send buffers for temporarily storing user data. When a socket is created, an INPCB is allocated, its fields are properly filled, and the INPCB is placed on the appropriate protocol's queue. When a socket is deleted, its INPCB is removed from the protocol queue.

3.1. Where to Put the Protocol

Our first decision was to determine where our protocol would fit into the existing protocol hierarchy. As mentioned previously, we implemented the protocol on top of UDP. We had to choose between this and putting it directly above IP. The option of developing the protocol directly above the raw network was quickly discarded since this would not allow any internet addressing. Nelson's [14] and Larus's [11] measurements dissuaded us from implementing the protocol above TCP [17]. They showed there is a penalty for using a byte stream protocol to implement a connectionless protocol. The choice of whether to implement the protocol above IP [16] or UDP was less obvious. UDP provides a datagram service that handles process-to-process addressing, but incurs the extra overhead of passing through UDP routines when sending or receiving data. IP does not have this extra layer of routines to go through, but we would have been required to duplicate most of the existing UDP code to handle process-to-process addressing. IP only handles host-to-host addressing. We decided to take advantage of the existing code and implemented the protocol on top of UDP.

What was not completely understood at the time of the decision was the inability to cleanly layer protocols on top of UDP or TCP [9]. This problem necessitated some hooks in the UDP code, which will be discussed later.

3.2. Interface to Socket Abstraction

The asymmetry of the RPC protocol posed a number of problems in fitting the protocol into the existing Berkeley UNIX socket implementation.

Looking at the existing protocols, it became apparent that each protocol was paired with exactly one socket type. Examples of this include the UDP protocol's support of a datagram socket type and the TCP protocol's support of a byte-stream socket type. The socket type is used to specify the semantics of the communication [12]. Since the currently supported protocols are symmetrical (connected sockets follow the same communication rules), one socket type is sufficient for describing the protocol. However, the RPC protocol is asymmetrical. The semantics of the protocol differ depending on whether a socket is used by a client or server. Since the IPC mechanism in 4.2BSD does not allow one protocol to implement more than one socket type, it became apparent that we could not interface to the mechanism using separate client and server socket types.

The thought of implementing two separate protocols, an RPC-client and RPC-server protocol, was not very appealing. Intuitively, RPC is one protocol - there is only one set of rules for communicating between client and server. Also, although the protocol is asymmetrical, the differences are slight, so most of the code would be duplicated if two separate protocols were implemented. Thus, since we could not implement one protocol with two socket types nor two protocols with two socket types, the only possibility was using one protocol with one socket type.

Once that decision was made, we needed some way to inform the protocol whether the socket was to be used by a client or server. One suggestion was to use the first *send* or *receive* system call on the socket as an indication of the socket type. The socket type would be set implicitly. However, with this approach, a user not fully understanding the protocol could accidentally initialize the socket to the wrong type by making the wrong system call. This was not "user-friendly". Thus, we decided to use the set socket option (*setsockopt*) capability that already existed [8]. The user must set the socket to **client** or **server** before sending or receiving data. Since the socket type is used to enforce the semantics of the protocol, setting the socket to type **client** allows us to force the client to send a request before receiving a response. Similarly, once the socket is set to type **server**, the server must receive data before trying to send it.

3.3. Segmentation Issues

Another problem that had to be addressed was how to handle multiple segment messages. We had to choose between making the user or the RPC protocol responsible for handling them. UDP imposes a fixed maximum datagram size, but call and return messages can be arbitrarily long. We first considered placing a 64K and then an 8K limit on the amount of data the user could send at once, but finally decided we did not want to restrict the user at all. Thus, some segmentation would have to be handled in user code due to the memory constraints of the kernel. If segmentation was only handled in the kernel, the available amount of memory in the kernel would place an upper bound on the amount of data the user could send. This is because our protocol keeps a copy of the data until it is reliably delivered.

After deciding that the responsibility of segmenting a call or return message would be solely the user's, we had to decide on a maximum segment size. We decided that we did not want IP to fragment our segments, because if IP fragments, it makes a copy of the data. This would mean that two copies of our data would be stored in the kernel (our protocol stores one copy of the data). Nelson emphasized the penalties of data copying [14]. Since the Berkeley UNIX implementation of IP fragments datagrams longer than 1536 bytes, we established a maximum segment size of 1K bytes. Some of the remaining bytes are used for the UDP header.

3.3.1. I/O Controls

After the maximum segment size was decided, we needed a method that would allow the user to inform the kernel whether multiple segment or single segment calls were being made. We

wanted to optimize the most common case - single segment messages. To do this, we treat a send or a receive as a single segment call or return. This is our default case. We decided to use I/O controls to distinguish the sending of multiple segment messages. An *ioctl* system call is used to indicate a multiple segment message will follow, and another *ioctl* system call is made before sending the last segment of a multiple segment message. These two calls simply clear or set the LAST_SEGMENT bit in the PCB, thereby affecting all subsequent segments sent. An example of this can be seen in Section 4. Although this method involves two extra system calls for each multiple segment message sent, such long messages are expected to occur infrequently. If many lengthy messages are to be sent, the user should choose another protocol.

Another possibility is to use two different send system calls. One would mean more segments were to follow, and the other would indicate the last segment was being sent. We did not consider a solution of this type since we wanted to use existing socket interfaces wherever possible. A similar approach was taken by Birrell at the DEC Systems Research Center [7].

Sending multiple segment messages proved to be more straightforward than receiving them. One approach to letting the user know how many segments to receive is simply to assume the recipient knows *a priori* how much data to expect. This is a reasonable assumption when using the protocol only as part of an RPC package (with a stub compiler). Both the client and server would then know the size of parameters and return data.

To make the protocol more flexible, we wanted a method to indicate to the user when there would be no more data. The best solution we could come up with was to use an error return as a way to report end of data. We store an error in the socket data structure when the last data segment is received. This error is returned to the user when the user tries to read and there is no more data. After an error is returned, it is removed from the socket data structure. Ideally, this would allow the recipient either to read until error, or to read the exact number of segments if this is known. Unfortunately, the existing socket mechanism forces the user to read until error in order to clear the socket error flag and thus allow subsequent sends to proceed without error. If there was a way to distinguish receive errors from send errors, this problem could easily be alleviated. Also, this solution forces the user to make an extra system call. This is another example of problems with the 4.2BSD socket implementation.

3.4. The Meaning of ACK

An acknowledgement of a segment could mean either that the remote kernel has received the data or that the data has been received and delivered to the remote user. We decided to use the former definition of ACK - just that the kernel has received the data, so the sender would not block until the following receive or send. To enforce flow control, we drop any subsequently sent data if the previous data has not been delivered to the user. Retransmissions ensure its final delivery. Note that implicit ACKs convey more information than explicit ACKs: implicit ACKs mean the data has been received by the remote user; not just the remote kernel.

An example will demonstrate how this works. Assuming a client sends the first segment of a multiple segment message, the client's call to send will return as soon as the message is sent. Upon receiving the segment, the server will send an acknowledgement. Now let us assume the client sends the next segment of the message before the server has delivered the last segment to the user. When the server receives the second segment, it will drop it because the last segment has not been read by the user. The server will acknowledge the first segment again. Since the second segment will not have been acknowledged when the retransmission timer expires, the client will retransmit the second segment. If the server has still not delivered the first segment to the user, the server will again drop the second segment and acknowledge the first segment. Since it is possible for this scenario to occur more than MAX_RETRY times (if the user on the server side is very slow in responding) we do not want to return an error to the client indicating there has been a break in communication when this has not occurred. So, in this case, when we receive an acknowledgement of the first segment after the second has been sent, we always reset our retry

counter to ensure we will not time out. Thus, the acknowledgement of the first segment is treated as if it were acknowledging a probe.

4. User Interface

In the first implementation of our protocol, a server could serve only one client at a time. The underlying protocol is the same whether or not there are multiple clients per server. Thus, we first designed the single client per server user interface and then extended the design to handle multiple clients per server.

Implementing our protocol using the socket abstraction allowed us easily to support all of the socket-related system calls [8]. After initial protocol testing, we realized there were still some remaining decisions to be made concerning the user interface, in addition to the *setsockopt* and *ioctl* issues discussed previously. Utilizing the existing system calls, we still had to place restrictions on the order in which they could be called and on which ones should and should not be supported. The following sections describe the justification for the restrictions we ultimately placed on the user.

To better understand these design decisions, it is best to describe our client/server model. We assume services are registered with a name server or may be found at some well-known address. Once the service is made available, the server waits until the service is requested. Upon receiving a request, the server forks a child process to handle the request, thus allowing the parent process to handle requests from other clients. The client merely sends a request to the server and waits for a response.

4.1. Single Client Per Server

With this model in mind, we originally thought the client would use the *sendto* system call. This was necessary because *sendto* includes the address of where the data is going. We needed the address since no connection is established prior to the first send. The server, in turn, would use *recvfrom* and then *sendto* so it could accept requests from any client without prior knowledge about the client's address.

We found this arrangement unacceptable after looking at the typical implementation of these system calls. We found that every *sendto* was actually establishing an internal connection (by storing the remotes address in the INPCB) and then tearing it down with each invocation of the system call. It would be more efficient to establish a permanent connection between client and server after the client's first send. By permanent connection we only mean storing the server's address in the client's INPCB and vice versa. Since the foreign address was permanently stored, there was no need to call *sendto* after the initial *sendto* from the client. The *send* system call would suffice. The problem now was that the user interface had become inconsistent. The first call from a client must be a *sendto*, but other calls could be either *send* or *sendto*. We did not like the idea of sometimes returning an error if *send* was used and sometimes allowing it. To solve this problem, we decided to force the client to make an explicit connection (by calling *connect*) before sending any data. Note that the protocol is still connectionless since calling *connect* does not result in any message transmissions.

The server, on the other hand, can never call *connect* because it does not know who its clients are until it receives a call from one. Thus, when a server receives a request from a new client, an implicit connection is established by storing the client's address in the server's INPCB. This allows the server also to use regular *send* and *receive* calls.

Neither the client nor the server sockets can be disconnected and subsequently reconnected; every client/server pair has unique sockets. When a socket is disconnected, its state is maintained for the duration of the no activity timer. This is done by the kernel independently of the user's program. Thus, if the user wanted, for example, to disconnect a client socket and then immediately reconnect it to another server, the connection could only be allowed if the request was made after the no activity timer expired. We could implement a blocking connect (suspending

the user process until a connection could be established), but we chose not to. Thus, we do not allow disconnects. This forces a client to use a separate socket for each server.

The decision is not difficult to change. If it proves more useful to allow disconnects, a blocking connect can easily be implemented.

An example of single client per server pseudo-code is given in Figure 5.

```
CLIENT                          SERVER

socket                          socket
setsockopt CLIENT               setsockopt SERVER
bind                            bind
connect to server               make socket name available
loop                                    to a name server
   snd()                        loop
   rcv()                           rcv()
end loop                           snd()
close                           end loop
                                close


snd
   procedure
      if multiple segment message
         setioctl MORE SEGMENTS
         loop
            send()      /* send all but last segment */
         end loop
         setioctl LAST SEGMENT
      end if
      send()            /* send last segment */
   end procedure

rcv
   procedure
      loop
         recv()
      end loop
   end procedure
```

Figure 5: Single Client per Server Pseudo-Code

4.2. Multiple Clients Per Server

The *listen* and *accept* system calls are used to handle multiple clients per server. The client code remains the same as the single client per server case. The **rcv and** snd routines are also the same. An example of multiple clients per server pseudo-code is illustrated in Figure 6. The following is a description of typical server code.

The server must create a socket, set it to type server, bind it to a local address, and then call *listen*. The *listen* call tells the system that the server is ready to queue up incoming calls. After calling *listen*, the server typically loops indefinitely, calling *accept* and forking off processes, implementing the service with the socket returned by *accept*. Since the server never knows when a client is completely finished with the service, we left it up to the user to close a given server socket after a reasonable period of time. This will not cause a problem for the client: if the client is not finished with the server, the client's next request will be accepted by the server loop and a

```
                                SERVER

                                socket
                                setsockopt SERVER
CLIENT                          bind
                                make socket name available
socket                                  to a name server
setsockopt CLIENT               listen
bind                            loop
connect to server                  accept connection from client
loop                               fork
   snd()                              loop
   rcv()                                 rcv()
end loop                                 snd()
close                                 end loop
                                      close
                                end loop


snd
   procedure
      if multiple segment message
         setioctl MORE SEGMENTS
         loop
            send()      /* send all but last segment */
         end loop
         setioctl LAST SEGMENT
      end if
      send()            /* send last segment */
   end procedure

rcv
   procedure
      loop
         recv()
      end loop
   end procedure
```

Figure 6: Multiple Clients per Server Pseudo-Code

new connection will be established on the server side. All of this will be transparent to the client and will cause no interruption of service.

5. The Complete RPC Package

In order to provide a complete RPC system, more than a transport level protocol is required. Cooper's Circus code is currently being modified to use our transport protocol rather than an inefficient user code implementation. Also, Cooper's stub compiler for the C programming language is being retargeted for our protocol, and his binding agent and runtime library are being modified accordingly. Our protocol is not tied to any particular binding agent; the DoD Internet name server under development at Berkeley could also be used [20,22].

6. Observations about the UNIX Interface

While designing and implementing our RPC protocol, we had the opportunity to examine closely the socket code and in some areas push it to its limits. Most of the difficulties encountered can be classified into two main categories: layering protocols and the symmetry expected in protocols. We also found the socket user interface rather restrictive and cumbersome. Most of these problems have already been addressed [18]. Discussion about the two main categories of problems follows.

6.1. Layering

During the initial design phase of our protocol, Mike Karels pointed out that clean interfaces did not exist to facilitate layering protocols in the kernel. After working with the code, we noted the following things that substantiated Karel's observations.

Only one protocol-specific protocol control block (PCB) can be attached to an internet PCB (INPCB). This implies that only one layer of protocols was expected to be implemented above IP. Fortunately, UDP does not need its own PCB since no state information is maintained, thus allowing us to attach our RPC PCB to the socket's INPCB. If we had chosen to implement a protocol above TCP, we would have had complications.

Similarly, both UDP and TCP store their data directly on the socket's receive buffer. No provisions were made for the possibility of switching to another input routine as is done at the IP level. The IP header contains a field indicating the next protocol the input data should be handed to. The UDP and TCP header have no similar field. This again suggests that the design does not facilitate the layering of protocols.

Furthermore, it is assumed that a socket's INPCB will be placed on a queue of INPCB's for one of the protocols implemented directly above IP. This can be seen in the protocol input routines. Each protocol input routine only searches its own protocol's queue of INPCB's for an address matching that of the incoming data. Thus, we were forced to place RPC socket INPCBs on the UDP queue instead of having a separate RPC INPCB queue. This meant that we often had to make a special case for the RPC socket type when the UDP queue was being searched. Examples of this follow. During socket abort, the RPC delete routine must be called for RPC sockets so that the RPC PCB can be deleted. The RPC timeout routine must check the entire UDP queue to locate RPC sockets. The UDP input routine must check the socket type to determine if it should simply store the data on the socket's receive buffer or send the data up to the RPC input routine. Each of these cases requires a special check to see whether a UDP INPCB is being used for an RPC socket.

6.2. Symmetrical Protocols

After implementing our protocol, we observed that many of our interfacing difficulties resulted from the fact that the IPC mechanism was designed with symmetrical protocols in mind. This became apparent when we were writing our connection establishment code and socket error reporting code, and when we were trying to find a way to distinguish between client and server

socket types. Although TCP connection establishment is asymmetrical (it involves active and passive sockets), TCP data transport is symmetrical.

The current IPC facilities expect a protocol to require a connection explicitly or to remain connectionless. There is no provision for requiring only one side of a protocol to make a connection. If a connection is required, a "connection required" attribute is associated with the protocol when it is defined. Once a protocol is characterized as "connection required", all necessary blocking and error returning is handled correctly if the user tries to send or receive on an unconnected socket. This is very nice, but we could not take advantage of it because we only require the client to be connected, not the server. Moreover, the server can only make an implicit connection. We needed the capability of attaching the "connection required" attribute to a socket type (client or server) as opposed to a protocol.

The problems encountered when trying to support two socket types with one protocol have already been discussed. They are mentioned again here to point out that we view them as asymmetry problems.

We found that allowing only one socket error to be stored per socket was too restrictive. The ability to store separate receive and send errors would have made our user interface much cleaner and more flexible. This was discussed earlier in Section 3.3.

7. Performance

Measurements were taken to determine the effect message length has on the speed of a remote procedure call. We measured the elapsed time of a remote procedure call [both call and return] with varying argument and return lengths. Two sets of tests were performed. The first set always returned 10 bytes of data while varying the argument length from 100 to 2100 bytes. In the second set of tests, the amount of data returned was always equal to the amount sent as the argument. The amount of data sent and returned ranged from 100 to 2106 bytes. The tests were performed on two VAX-11/750s* running in single-user mode, connected by a 10 megabit per second Ethernet cable.

The tests consisted of making 100 remote procedure calls with the specified argument and result lengths. The elapsed time, user CPU time and kernel CPU time were measured for each call and averaged over all of the calls. The elapsed time was determined by calling **gettimeofday** before and after each RPC, taking the difference, and averaging the difference for the 100 calls. The user CPU time and kernel CPU time were similarly obtained by calling **getrusage** before and after each RPC.

Table 1 presents the results from the first set of tests: varying argument lengths, constant return length. A graph of elapsed time verses argument length is shown in Figure 7. Table 2 presents the results from the second set of tests: varying argument and return sizes. Figure 8 shows a graph of elapsed time verses argument and return length for the data in Table 2.

The measurements suggest that the elapsed time of a remote procedure call increases with increasing amounts of data being sent or returned. The decrease in elapsed time seen when 1K and 2K bytes are sent can be attributed to mbuf (memory buffer) usage. There are two sizes of mbufs: 128 bytes and 1024 bytes. 1024 byte mbufs are only used if the amount of data sent is greater than or equal to 1024. It is faster to use one 1024 byte mbuf than to use many smaller mbufs, which must be done when sending smaller amounts of data. The longer round trip times of remote procedure calls for data lengths greater than 1024 bytes is expected since more segments must be sent and acknowledged when multiple segment messages are being sent.

Our measurements suggest that the performance of RPC is good when sending small quantities of data and that its performance drops as larger amounts of data are sent. Recall that our protocol optimizes the single segment case.

*VAX is a trademark of Digital Equipment Corporation

arg. size (bytes)	result size (bytes)	elapsed time (msecs/rpc)	total cpu time (msecs/rpc)	user cpu time (msecs/rpc)	kernel cpu time (msecs/rpc)
100	10	17.4	7.0	1.1	5.9
200	10	20.7	8.9	0.5	8.4
300	10	22.8	10.1	0.7	9.3
400	10	23.3	9.4	0.7	8.7
500	10	25.9	11.6	0.5	11.1
600	10	26.8	11.4	0.5	10.9
700	10	28.6	12.5	0.7	11.8
800	10	30.4	12.4	0.6	11.8
900	10	31.1	13.1	0.5	12.6
1024	10	31.1	14.4	0.9	13.5
1280	10	37.7	17.6	1.4	16.2
1536	10	42.6	20.2	1.2	19.0
1792	10	47.3	21.4	0.8	20.6
2048	10	47.6	23.7	1.3	22.4
2100	10	52.5	23.7	0.9	22.8

Table 1: RPC performance with varying argument size, constant return

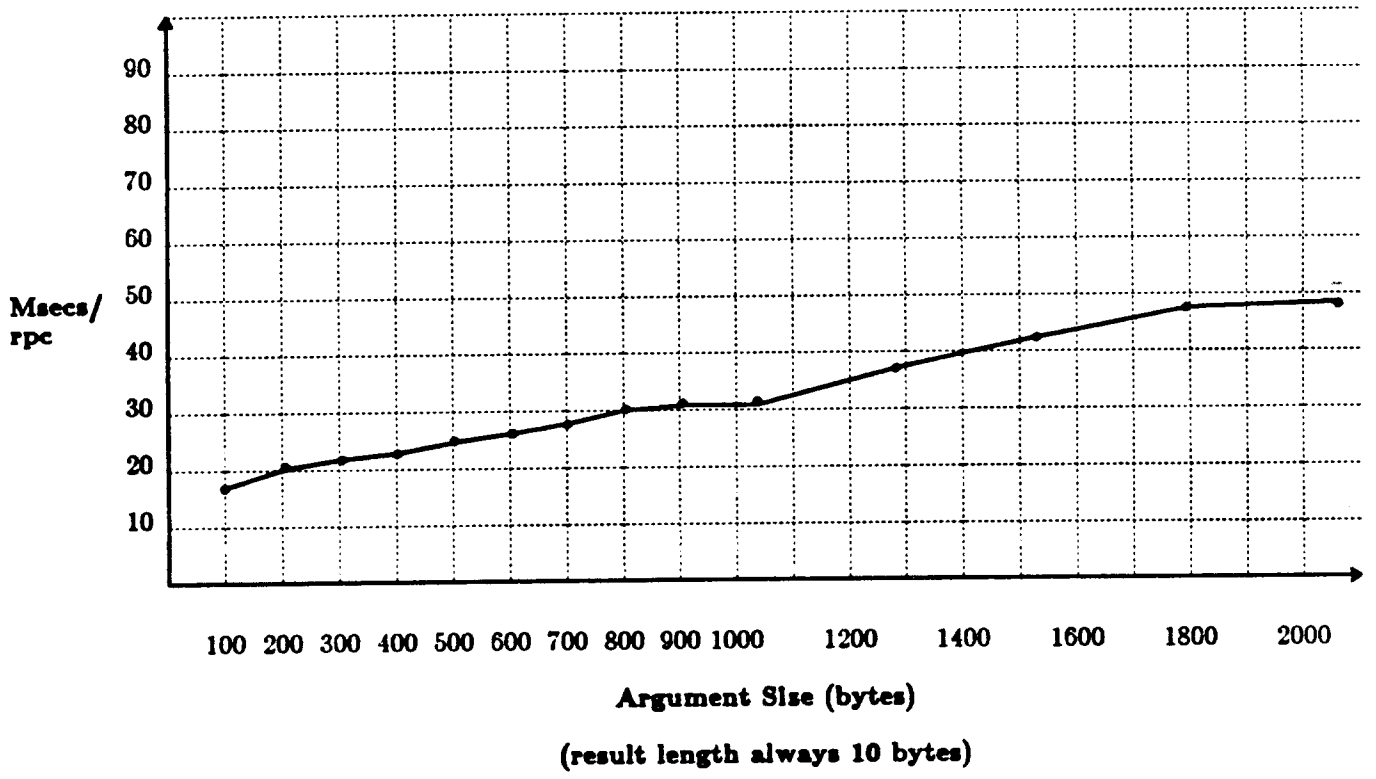

Figure 7: Elapsed time verses argument size (constant return)

arg. size (bytes)	result size (bytes)	elapsed time (msecs/rpc)	total cpu time (msecs/rpc)	user cpu time (msecs/rpc)	kernel cpu time (msecs/rpc)
100	100	19.9	7.9	0.5	7.4
200	200	22.8	10.2	0.7	9.5
300	300	25.4	11.3	0.5	10.8
400	400	29.0	10.9	0.2	10.7
500	500	32.1	12.5	0.9	11.6
600	600	35.6	13.9	0.9	13.0
700	700	39.5	12.1	0.5	11.6
800	800	41.6	15.1	0.9	14.2
900	900	44.9	16.3	1.3	15.0
1024	1024	38.0	16.1	1.5	14.6
1280	1280	59.2	27.8	0.9	26.9
1506	1536	66.8	24.2	1.1	23.1
1792	1792	75.6	26.5	1.5	25.0
2048	2048	72.7	27.8	1.0	26.8

Table 2: RPC performance with varying argument and result sizes

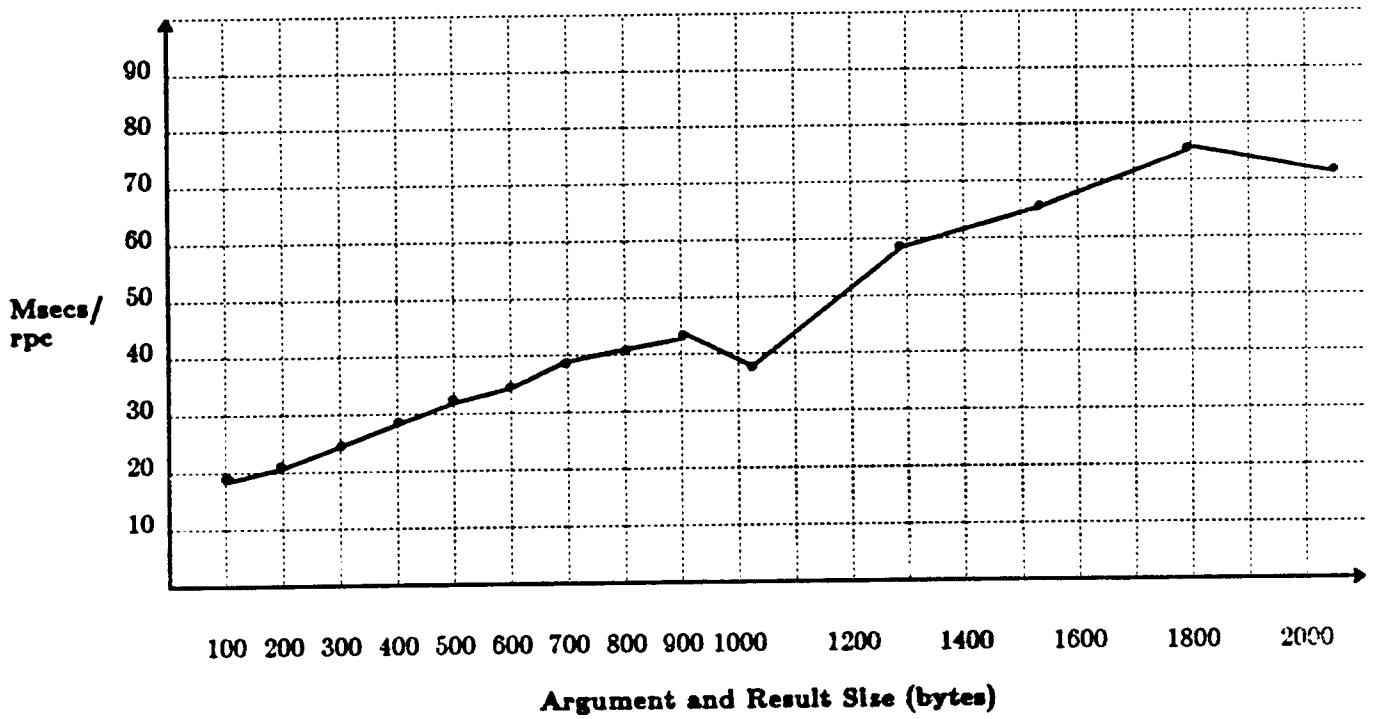

Figure 8: Elapsed time verses argument and result size

Table 3 compares our RPC implementation with Cooper's TCP-based Courier [11], Larus's Corrected PCourier [11], Circus, UDP, and TCP [6]. The UDP, TCP and Circus measurements were obtained from Cooper's work [6]. Recall that Cooper measured the time for an exchange of UDP datagrams, the time for an exchange of messages over a TCP byte-stream, and the time for an unreplicated Circus remote procedure call. A small number of bytes were exchanged for these measurements. The Courier and Corrected PCourier measurements were obtained from Larus's work [11]. Recall that these measurements were obtained by sending a four byte argument and returning a single integer. The RPC and RPC (no rexmit) measurements were obtained by sending twelve bytes and returning six bytes. The system clock used for the RPC measurements had a resolution of 10 milliseconds. Note that the RPC measurements were taken when running in single-user mode while the Circus, UDP, and TCP measurements were not. Also, Cooper's UDP test used additional system calls to time out **recv** calls (to recover from lost datagrams). We attribute the faster execution of our RPC implementation than the UDP test to these two factors. The RPC (no rexmit) measurement probably had no retransmisions (no lost packets), while the RPC measurement did. This explains the difference in the two measurements.

Our measurements suggest that our protocol is as fast as other kernel protocols and two to three times faster than other user-level code implementations.

protocol	elapsed time (msec/rpc)
UDP	26.5
TCP	23.2
RPC (no rexmit)	17.8
RPC	22.05
Circus	48.0
Courier	69.7
Corrected PCourier	25.9

Table 3: Performance Comparisons

8. Applications to Distributed Systems

RPC was developed to facilitate the design of distributed systems. It allows for rapid prototyping of any client/server system. Not only can distributed systems be developed quickly, they are also more likely to be correct since the communication code is hidden from the programmer.

The proliferation of personal workstations connected by a local area network is necessitating the development of centralized file servers, name servers, and mail servers [2]. This type of service is ideally suited to RPC.

9. Future Extensions

A few of the possible extensions that may be made to our protocol are discussed in this section.

One possible extension to our protocol is to add security and authentication. Applications often need to be certain of the identity of clients before granting them service. It may also be desirable to prevent eavesdropping on client-server sessions. No provisions were made for security or authentication in our implementation. In the present design, as in most transport protocols, it is the responsibility of the user to enforce security at another level. The design of a secure remote procedure call protocol is given by Birrell in [1].

Another possible extension is to incorporate broadcasting of messages. Cooper discusses this form of replicated procedure calls in [6]. Replicated procedure calls can be used to provide fault-tolerant distributed computing.

In addition, it is desirable to optimize calls made between clients and servers residing on the same machine. Currently there are no special provisions for loop-back.

Note that our protocol already contains the mechanisms required for a reliable datagram protocol. Recall that a reliable datagram protocol guarantees the reliable delivery of a single variable-length message in one direction. The state information and timers needed for guaranteeing reliability already exist. We would only have to provide the sending side of the protocol, and require every message to be explicitly acknowledged. This type of protocol would be useful for applications that expect information to be sent to them periodically, such as statistics gathering programs.

10. Conclusions

We have implemented an RPC protocol for the Berkeley UNIX kernel. The protocol has been retrofitted into Cooper's Circus stub compiler, binding agent and runtime library; the result is a complete, high performance RPC package. The protocol is two to three times faster than Cooper's TCP-based version of Courier and approximately twice as fast as his UDP-based Circus. The speed of the protocol itself is comparable to the speed of other kernel protocols such as UDP and TCP.

The two major difficulties encountered in integrating our protocol into the existing socket interface were layering the protocol and implementing an asymmetrical protocol in an environment geared for symmetry. Solutions to the first problem were found by using UDP INPCBs instead of having RPC INPCBs, and adding some hooks into the UDP code. The second problem was resolved by using I/O controls to distinguish client and sever socket types.

The main contribution of this work is that it provides a new alternative for constructing distributed applications under Berkeley UNIX. Since it is as efficient as the other kernel transport protocols, users will be free to choose the mechanism that best fits the semantics of their applications.

References

[1] Birrell, A.D. Secure Communication Using Remote Procedure Calls. *ACM Transactions on Computer Systems*, Volume 3, No. 1, February 1985, pp. 1-14.

[2] Birrell, A.D., Levin, R., Needham, R.M., and Schroeder, M.D. Grapevine: An exercise in distributed computing. *Communications of the ACM* 25(4), April 1982, pp. 260-274.

[3] Birrell, A.D., and Nelson, B.J. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, Volume 2, No. 1, February 1984, pp. 39-59.

[4] Cooper, Eric. Replicated Procedure Call. *Proceedings of the 3rd ACM Symposium on Principles of Distributed Systems*, August 1984.

[5] Cooper, Eric. Circus: A Replicated Procedure Call Facility. *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems*, October 1984, pp. 11-24.

[6] Cooper, Eric. *Replicated Distributed Programs*. Ph.D. dissertation, Computer Science Division, University of California, Berkeley, April 1985.

[7] Cooper, Eric. Private communications, 1985.

[8] Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, K., and Mosher, D. *4.2BSD System Manual*. Computer Systems Research Group, Computer Science Division, University of California, Berkeley, July 1983.

[9] Karels, Mike. Private communications, 1984.

[10] Larus, J. On the Performance of Courier Remote Procedure Calls Under 4.1c BSD. Report No. UCB/CSD 83/123, University of California, Berkeley, August 1983.

[11] Kupfer, M. Performance of a Remote Instrumentation Program. Report No. UCB/CSD 85/223, Computer Science Division (EECS) University of California, Berkeley, February 1985.

[12] Leffler, S.J., Fabry, R.S., and Joy, W.N. A 4.2BSD Interprocess Communication Primer. Computer Systems Research Group, Computer Science Division, University of California, Berkeley, 1983.

[13] Leffler, S.J., Joy, W.N., and Fabry, R.S. 4.2BSD Networking Implementation Notes Revised July, 1983. Computer Systems Research Group, Computer Science Division, University of California, Berkeley, 1983.

[14] Nelson, Bruce Jay. *Remote Procedure Call*. Ph.D. dissertation, Computer Science Department, Carnegie-Mellon University, CMU report number CMU-CS-81-119, Xerox PARC report number CSL-81-9, May 1981.

[15] Postel, Jon. *User Datagram Protocol*. Information Sciences Institute, University of Southern California, RFC 768, August 1980.

[16] Postel, Jon. *Internet Protocol*. Information Sciences Institute, University of Southern California, RFC 791, September 1981.

[17] Postel, Jon. *Transmission Control Protocol*. Information Sciences Institute, University of Southern California, RFC 793, September 1981.

[18] Sechrest, S. Tutorial Examples of Interprocess Communication in Berkeley UNIX 4.2BSD. Report No. UCB/CSD 84/191, University of California, Berkeley, June 1984.

[19] Sun Microsystems. Remote Procedure Call Reference Manual. Mountain View, California, October 1984.

[20] Terry, D.B., Painter, M., Riggle, D.W., Zhou, S. The Berkeley Internet Name Domain Server. Report No. UCB/CSD 84/182, Unversity of California, Berkeley, May 1984.

[21] Xerox Corporation. *Courier: The Remote Procedure Call Protocol.* Xerox System Integration Standard XSIS-038112, December 1981.

[22] Zhou, S. The Design and Implementation of the Berkeley Internet Name Domain (BIND) Servers. Report No. UCB/CSD 84/177, University of California, Berkeley, May 1984.