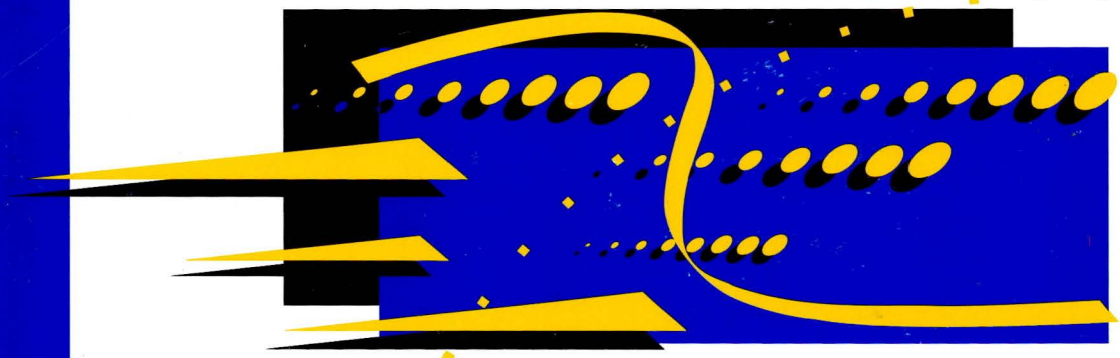


CONVEX

Network Programming Guide

First Edition

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

CONVEX Network Programming Guide

Order No. DSW-106

First Edition
February 1994

CONVEX Press
Richardson, Texas
United States of America

CONVEX Network Programming Guide

Order No. DSW-106

This entire book is recyclable.

Printed in the United States of America

Revision information for CONVEX Network Programming Guide

Edition	Document No.	Description
First	710-023730-000	Initial release February, 1994 with ConvexOS V11.0. Replaces the following documents: • *CONVEX Interprocess Communication (IPC) Programming Guide* • *CONVEX TLI Library Programmer's Guide* • *CONVEX NFS Reference Set*

Contents

Figures

Tables

Preface

Purpose and audience

The *CONVEX Network Programming Guide* provides experienced C language systems programmers with information needed to write networking applications for CONVEX systems.

This guide describes network programming concepts and techniques for each application program interface on a CONVEX system. Example code in this guide illustrates basic and advanced use of system calls, library routines, or data structures for each interface.

Prerequistites to use of this guide include familiarity with basic networking concepts and the C language, as well as significant programming experience in a ConvexOS environment.

Organization

This guide is divided into four parts, one for each interface:

- Part 1—Sockets interface
- Part 2 —STREAMS interface
- Part 3—Transport Layer Interface (TLI) library
- Part 4—Network File System (NFS) interface

Notational conventions

This section discusses notational conventions used in this book.

Command syntax

Consider this example:

```
COMMAND input_file [...] {a | b} [output_file]
   ①        ②      ③     ④       ⑤
```

1. `COMMAND` must be typed as it appears.
2. *input_file* indicates a file name that must be supplied by the user.
3. The horizontal ellipsis in brackets indicates that additional input file names may be supplied.
4. Either `a` or `b` must be supplied.
5. *[output_file]* enclosed in brackets indicates an optional file name supplied by the user.

General conventions

In general, the following conventions are used in this guide:

- Italics designate:
  - User-supplied variables in a command-line example
  - New and important terms
  - Variables in mathematical equations
  - Document titles
- `Constant-width font` designates:
  - System output in screens and examples
  - Command names and options
  - System calls
  - Data structures and types
  - Directives, program statements, display examples, printout examples, and error messages returned
- **`Bold, constant-width font`** designates user input in screens and examples, and must be typed exactly as it appears.
- Horizontal ellipsis (...) shows repetition of the preceding item(s).

- Vertical ellipsis shows that lines of code have been left out of an example.
- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **RETURN** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-X** indicates that you must press and hold down the **CTRL** key and then press the **X** key.
- The word "enter" in a phrase such as "enter `ls`" means that you type the command and then press **RETURN**.
- References to the *ConvexOS Programmer's Reference* appear in the form adb(1), where the name of the man page is followed by its section number enclosed in parentheses.

Note

A note highlights supplemental information.

Caution

A caution highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Associated documents

Using this software may require information not specific to the tasks described in this document.

For more information on the ConvexOS operating system, Internet Services, and NFS, refer to the following CONVEX documentation:

- Man pages sections 2 through 5 for the ConvexOS operating system, Internet Services, and NFS.
- *Managing CONVEX Internet Services and NFS*, DSW-108. This book contains information about administering systems on a CONVEX TCP/IP network and explains how to configure and run NFS/NIS.

For more information on STREAMS and networking interfaces, refer to the following documents:

- AT&T, *UNIX System V Release 4.0 Programmer's Guide: Networking Interfaces*, Prentice Hall, Englewood Cliffs, N.J., 1990
- AT&T, *UNIX System V Release 4.0 Programmer's Guide: STREAMS*, Prentice Hall, Englewood Cliffs, N.J., 1990
- AT&T, *UNIX System V Interface Definition, Volume 1, Third Edition*

Ordering documentation

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Include the order number or the exact title, as listed on the front cover.

Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call 1(800)952-0379.
- From Canada, call 1(800)345-2384
- Outside continental U.S., contact local CONVEX office.

The `contact` utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` mails it to the TAC electronically. The TAC notifies you within 48 hours that your report has been received.

To use `contact` requires:

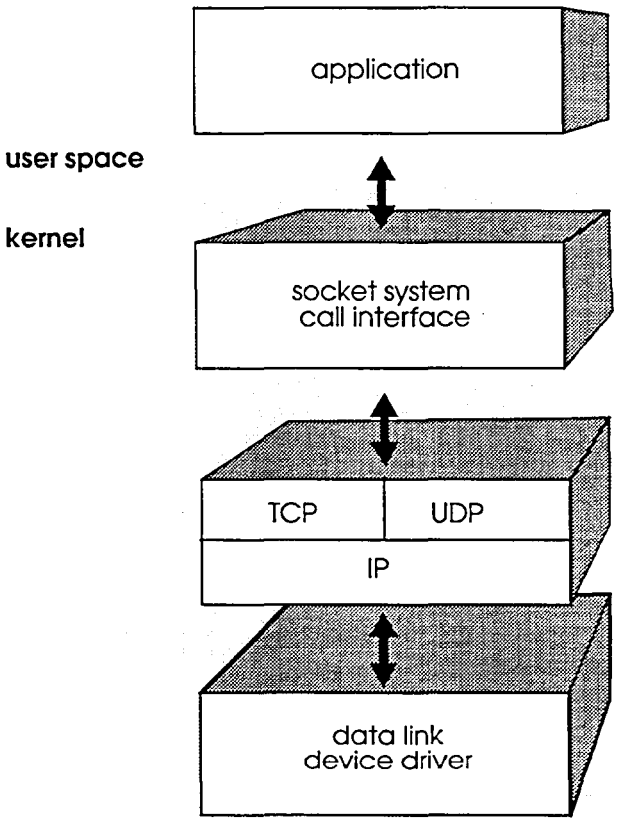
- UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC.
- Full path name of the program or utility in question.
- Version number of the program or utility in question.

Refer to the contact(1) man page for complete details.

Acknowledgments

The authors wish to thank all the people at CONVEX and the users who contributed their ideas and time in the development of this book.

Sockets interface



Sockets interface

This part explains how to use the Berkley socket interface for network programming applications.

This part is organized into the following chapters:

- Chapter 1 introduces socket programming and describes the basic use of network library routines.
- Chapter 2 describes intermediate socket programming techniques, including how to create datagrams and establish connections.
- Chapter 3 discusses a variety of advanced socket programming techniques, including the use of connectionless sockets and the Internet "superserver" daemon, `inetd`.

Basic socket programming 1

Introduction

Processes on the same machine can communicate using pipes, socket pairs, named pipes, and sockets. When communicating between processes on different machines, you need to have two processes separately create and name sockets to establish a connection between the processes. The socket facility enables you to do this.

This chapter provides a high-level description of socket facilities provided with ConvexOS. It is designed to complement man pages for IPC primitives with examples of their use. The remainder of this chapter is organized into four sections:

- Understanding sockets, domains, and socket types.
- Creating, using, and discarding sockets.
- Using network library routines to acquire remote addresses and information needed for socket programming.
- Constructing a typical distributed application.

ConvexOS IPC facilities support two address families and allow processes to rendezvous in many ways. Processes may rendezvous through a file-system-like name space (a space where all names are path names) as well as through a network name space. Further, communication facilities have been extended to include more than the simple byte stream provided by a pipe.

Understanding sockets

The basic mechanism for interprocess communication is the *socket*. A socket is an endpoint of communication to which a name may be associated or *bound*. Each socket in use has a type and one or more associated processes. (Socket types are explained in more detail subsequently.)

Domains

Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to named sockets. For example, in the UNIX communication domain, sockets are named with path names; for example, a socket may be named /dev/foo. Sockets exchange data only with sockets in the same domain. ConvexOS IPC facilities support two communication domains: the UNIX domain, for local communication, and the Internet domain, for use by processes that communicate using standard Internet communication protocols. The Internet domain can be used both locally and remotely.

Underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as on the interface to socket facilities available to a user. An example of the latter is that a socket operating in the UNIX domain sees a subset of error conditions possible when operating in the Internet domain.

Socket types

Sockets are typed according to communication properties visible to a user. Processes communicate only between sockets of the same type. ConvexOS IPC facilities support two types of sockets:

- **Stream sockets**—provide for bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes. (In the UNIX domain, in fact, the semantics are identical and, as you might expect, pipes have been implemented internally as simply a pair of connected stream sockets.)
- **Datagram sockets**—support bidirectional flow of data that is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages lost or duplicated, and, possibly, in an

order different from the order in which they were sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model facilities found in many contemporary packet-switched networks such as Ethernet.

Note

The raw socket interface to the IP protocol is no longer supported. ConvexOS now provides a STREAMS interface to IP conforming to the NPI (Network Provider Interface) standard. For information on converting raw socket applications to STREAMS and NPI, refer to Chapter 7, "Converting raw socket applications to STREAMS" on page 141.

Using socket system calls

Installations often use multiple network interfaces; for example, Ethernet for normal use and UltraNet for high-speed file transfers. The CONVEX UltraNet Interface uses its own socket system calls; other CONVEX network interfaces use standard ConvexOS socket calls. To use socket system calls—*socket, bind, accept, connect, listen,* etc.—network application programs are linked with either standard socket system calls in `libc.a` or with the UltraNet Sockets Compatibility Library, `libulsock.a`. By default, programs are linked with `libc.a`; for a program to use UltraNet sockets, you must specify `libulsock.a` with the `-l` option when you compile it. For example, if you enter the command:

```
cc prog.c -lulsock
```

`prog.c` is linked with the UltraNet Sockets Compatibility Library.

The CONVEX UltraNet Interface includes both a *native* UltraNet interface and an Internet interface module that allows it to participate in a TCP/IP network. When a program is linked with UltraNet sockets, socket-related calls are routed to the appropriate network interface. CONVEX UltraNet software uses the destination host address to determine the type of socket you need (TCP/IP or UltraNet) for the path you want to take. Refer to *CONVEX Networking Concepts* for more information about the CONVEX UltraNet Interface.

Creating sockets

To create a socket, use the `socket` system call. This call requests the system to create a socket in the specified domain and of the specified type. The syntax of the command is

```
s = socket(domain, type, protocol);
```

where:

s Descriptor, in the form of a small integer number, returned by the system for use in later system calls that operate on the socket.

domain One of the manifest constants defined in the <sys/socket.h> file. For the UNIX domain, the constant is `AF_UNIX`; for the Internet domain, it is `AF_INET`. Domain manifest constants are named `AF_`*domain* to indicate the "address family" to use in interpreting names in that domain.

type One of the manifest constants: `SOCK_STREAM` or `SOCK_DGRAM`.

protocol This argument is currently ignored.

To create a stream socket in the Internet domain, use a call similar to the following:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call results in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for use on the local system, use a call similar to the following:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

There are several reasons a `socket` call may fail. Aside from rare occurrence of lack of memory (`ENOBUFS`), a socket request may fail due to a request for an unknown protocol (`EPROTONOSUPPORT`), or a request for a type of socket for which there is no supporting protocol (`EPROTOTYPE`).

Binding names to sockets

When you create a socket, it has no name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet domain, an association is composed of local and foreign addresses, and local and foreign ports; while in the UNIX domain, an association is composed of local and foreign path names. The phrase "foreign path name" means a path name created by a foreign process, not a path name on a foreign system.

In most domains, associations must be unique. In the Internet domain, there may never be duplicates of the ordered set < local address, local port, foreign address, foreign port>. UNIX domain sockets need not always be bound to a name, but when bound, there may never be duplicate <local path name, foreign path name> sets. Path names may not refer to files already existing on the system.

The `bind` system call allows a process to specify half of an association, <local address, local port> or <local path name>, while the `connect` and `accept` primitives are used to complete a socket's association.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because `connect` and `send` calls automatically bind an appropriate address if they are used with an unbound socket.

The syntax of the `bind` system call is:

`bind`(*s, name, namelen*);

where:

s	Descriptor returned by the `socket` system call.
name	Variable-length byte string, which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain, as this is one of the properties that constitute the domain. In the Internet domain, names contain an Internet address and port number. In the UNIX domain, names contain a path name and a family, which is always `AF_UNIX`.
namelen	Length of the string that contains the name.

The example in Figure 1 binds the name, `/tmp/foo`, to a UNIX domain socket.

Figure 1 Binding a UNIX domain name to a socket

```
#include <sys/un.h>
        .
        .
        .
struct sockaddr_un addr;
        .
        .
        .
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
       sizeof (addr.sun_family));
```

The `sockaddr_un` structure shown in Figure 1 is used in the UNIX domain to store domain names and path names. In the Internet domain, host address, port numbers, and domain names are carried in the `sockaddr_in` structure. You use `getservbyname` to fill in the port number field. Refer to the getservbyname(3N) man page for details.

In determining the size of a UNIX domain address, null bytes are not counted, which is why `strlen` is used. In the current implementation of UNIX domain IPC, the file name referred to in `addr.sun_path` is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where `addr.sun_path` is to reside, and this file should be deleted by the caller when it is no longer needed.

Binding an Internet address is more complicated. The actual call is similar, as shown in Figure 2.

Figure 2 Binding an Internet address to a socket

```
#include <sys/types.h>
#include <netinet/in.h>
        .
        .
        .
struct sockaddr_in sin;
        .
        .
        .
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

but the selection of what to place in address, `sin`, requires discussion. The problem of formulating Internet addresses is described later in this chapter in the section, "Using network library routines," which discusses library routines used in name resolution.

Initiating socket connections

Connection establishment is usually asymmetrical, with one process acting as a *client* and the other performing the role of a *server*. When willing to offer its advertised services, the server binds a socket to a well-known address associated with the service and then passively listens on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a connection to the server's socket. On the client side, the `connect` call is used to initiate a connection. Using the UNIX domain, this system call might appear as shown in Figure 3.

Figure 3 Initiating a connection to a UNIX domain socket

```
struct sockaddr_un server;
        .
        .
        .
connect(s, (struct sockaddr *)&server,
       strlen(server.sun_path) + sizeof (server.sun_family));
```

Figure 4 shows a `connect` system call used to initiate a connection to an Internet domain socket.

Figure 4 Initiating a connection to an Internet domain socket

```
struct sockaddr_in server;
          .
          .
          .
connect(s, (struct sockaddr *)&server, sizeof (server));
```

In Figure 3 and Figure 4, `server` contains either the path name or the Internet address and port number of the server to which the client process wishes to connect. If the client process' socket is unbound at the time of the `connect` call, the system automatically selects and binds a name to the socket, if necessary; refer to Chapter 3, "Advanced socket programming." This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful; any name automatically bound by the system, however, remains. Otherwise, the socket is associated with the server and data transfer may begin.

Some of the more common errors returned when a connection attempt fails are:

`ETIMEDOUT` After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt. This usually occurs because the destination host is down, or because problems in the network resulted in lost transmissions.

`ECONNREFUSED` The host refused service for some reason. This is usually due to a server process not being present at the requested name.

`EISCONN` The socket is already connected.

`ENETDOWN` or `EHOSTDOWN`
These operational errors are returned based on status information delivered to the client host by underlying communication services.

`ENETUNREACH` or `EHOSTUNREACH`
These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or packet-

switching nodes. Many times the status returned is not sufficient to distinguish a failed network from a failed host, in which case the system indicates that the entire network is unreachable.

The server willing to receive a client's connection must perform two steps after binding its socket:

1. Be willing to listen for incoming connection requests. For example,

   ```
   listen(s, 5);
   ```

   The second parameter to the `listen` call specifies the maximum number of outstanding connections that may be queued awaiting acceptance by the server process; this number is currently limited to 5. Should a connection be requested while the queue is full, the connection is not refused, but rather the individual messages that constitute the request are ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. It is still possible to get the ETIMEDOUT error back, although this is unlikely. The backlog limit of 5 avoids the problem of processes monopolizing system resources by setting an infinite backlog, then ignoring all connection requests.

2. With a socket marked as listening, a server may then `accept` a connection. Figure 5 shows an example of code used to accept an Internet domain connection.

Figure 5 Server accepting a connection

```
struct sockaddr_in from;
          .
          .
          .
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

For the UNIX domain, `from` is declared as a `sockaddr_un` structure, but nothing different needs to be done as far as `fromlen` is concerned. The examples that follow discuss only Internet routines.

A new descriptor, along with a new socket, is returned on receipt of a connection. If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter, `fromlen`, is initialized by the server to indicate how much space is associated with `from`, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

`accept` normally blocks; that is, `accept` does not return until a connection is available or the system call is interrupted by a signal to the process. Furthermore, there is no way for a process to indicate it will accept connections from only a specific individual or individuals. It is up to the user process to consider who the connection is from and to close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the `accept` call, there are alternatives; they are considered in Chapter 3, "Advanced socket programming."

Transferring data

With a connection established, data may begin to flow. Several possible calls can be used to send and receive data. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect in this case, normal `read` and `write` system calls are then usable. For example, you might use commands similar to the following:

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to `read` and `write`, you can use `send` and `recv`.

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

While `send` and `recv` are virtually identical to `read` and `write`, the extra `flags` argument is important. The flags, defined in the <sys/socket.h> file, may be specified as nonzero values if one or more of the following options is required:

`MSG_OOB`	Send/receive out of band data.
`MSG_PEEK`	Look at data without reading.

Out-of-band data, a concept specific to stream sockets in the Internet domain, is discussed inChapter 3, "Advanced socket programming."

When MSG_PEEK is specified with a `recv` call, any data present is returned to the user, but treated as still unread; that is, the next `read` or `recv` call applied to the socket returns the data previously previewed.

Discarding sockets

Once a socket is no longer needed, discard it by applying a `close` to the descriptor, as follows:

```
close(s);
```

If data is associated with a socket that promises reliable delivery (for example, a stream socket) when a `close` takes place, the system continues to attempt to transfer data. If the data is still undelivered after a fairly long period of time, however, it is discarded. Should a user have no use for any pending data, the system may perform a `shutdown` on the socket prior to closing it. The `shutdown` system call is of the form:

```
shutdown(s, how);
```

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

Additional information

Many examples presented here can serve as models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create processes and communication paths. After this code is debugged, code specific to the application can be added.

You can find detailed information about particular calls and protocols in the ConvexOS man pages. The man pages listed in Table 1 are particularly relevant.

Table 1 Relevant man pages

For information about:	Refer to these man pages:
Internet addresses	inet(3N), inet(4F)
Creating and naming sockets	socket(2), bind(2)
Establishing connections	listen(2), accept(2), connect(2)
Host database	gethostent(3N), hosts(5)
Protocol database	getprotoent(3N), protocols(5)
Network services database	getservent(3N), services(5)
Transferring data	read(2), write(2), send(2), recv(2)
TCP/IP protocols	ip(4P), tcp(4P), udp(4P)

Using network library routines

This section considers C runtime routines provided to manipulate network addresses.

Locating a service on a remote host requires many levels of mapping before client and server can communicate. A service is assigned a name intended for human understanding; for example, "the login server on host monet." This name and the name of the peer host must then be translated into network addresses, which are not necessarily comprehensible to humans. Finally, the address must then be used to locate a physical location and route to the service.

The specifics of these mappings are likely to vary between network architectures. For instance, it is better if a network does not require hosts to use names that reveal their physical location to the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location-independent manner may induce overhead in connection establishment, because a discovery process must take place, but it allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and appropriate protocol to use in communicating with the server process. The <netdb.h> file must be included when using any of these routines.

Host names

An Internet host name-to-address mapping is represented by the `hostent` structure shown in Figure 6.

Figure 6 `hostent` structure

```
struct  hostent {
        char    *h_name;   /* official name of host */
        char    **h_aliases; /* alias list */
        int     h_addrtype; /* host address type
        int     h_length; /* length of address */
        char    **h_addr_list; /* addresses */
#define h_addr  h_addr_list[0] /* backward compatible */
];
```

The `gethostbyname` routine takes an Internet host name and returns a `hostent` structure, whereas the `gethostbyaddr` routine maps Internet host addresses into a `hostent` structure.

The official name of the host and its public aliases, along with the address type (family) and the address, are returned by these routines. The database for these calls is provided by the hosts file. For more information, refer to the hosts(5) man page.

Network names

Routines for mapping network names to numbers are also provided. These routines return a `netent` structure, as shown in Figure 7.

Figure 7 netent structure

```
struct  netent {
        char      *n_name;  /* official name of net */
        char      **n_aliases; /* alias list */
        int       n_addrtype; /* net address type */
        unsigned  long n_net; /* net no., host byte order */
};
```

The getnetbyname, getnetbynumber, and getnetent routines are the network counterparts to the host routines described in the previous section. These routines extract their information from the networks file. For more information, refer to the networks(5) man page.

Protocol names

The protoent structure shown in Figure 8 defines the protocol-name mapping used with the getprotobyname, getprotobynumber, and getprotoent routines.

Figure 8 protoent structure

```
struct  protoent {
        char      *p_name;  /* official protocol name */
        char      **p_aliases;/* alias list */
        int       p_proto;  /* protocol number */
};
```

These routines obtain their information from the protocols file. For more information, refer to the protocols(5) man page.

Service names

Information regarding services is slightly more complicated. A service is expected to reside at a specific port and to employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Furthermore, a service may reside on multiple ports. If this occurs, higher-level library routines must be bypassed or extended. Available services are listed in the services file. Service mapping is described by the servent structure shown in Figure 9.

Figure 9 servent structure

```
struct servent {
        char    *s_name; /* official service name */
        char    **s_aliases;/* alias list */
        int     s_port;  /* port number, net byte order */
        char    *s_proto; /* protocol to use */
};
```

The getservbyname routine maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus the call:

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol, whereas the call:

```
sp = getservbyname("telnet", "tcp");
```

returns only the service specification for the telnet server that uses the TCP protocol. The getservbyport and getservent routines are also provided. getservbyport has an interface similar to that provided by getservbyname; an optional protocol name may be specified to qualify lookups.

Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed in a network-independent fashion. It is clear, however, purging all network dependencies is very difficult. As long as the user is required to supply network addresses when naming services and sockets, there is always some network dependency in a program. The example client program shown in Figure 11 illustrates this point. (This example is considered in more detail subsequently.)

To make the remote login program independent of the Internet protocols and addressing scheme, you must add a layer of routines that mask network- dependent aspects from mainstream login code. For current facilities available in the system, this does not appear to be worthwhile.

Aside from address-related database routines, several other routines of interest are available in the runtime library. These routines are intended mostly to simplify manipulation of names and addresses. Table 2 summarizes routines for manipulating variable-length byte strings and handling byte-swapping of network addresses and values.

Table 2 Byte-handling routines

Call	Synopsis
`bcmp(s1, s2, n)`	Compare byte-strings; 0 if same, not 0 otherwise
`bcopy(s1, s2, n)`	Copy n bytes from s1 to s2
`bzero(base, n)`	Zero-fill n bytes starting at base
`htonl(val)`	Convert 32-bit quantity from host to network byte order
`htons(val)`	Convert 16-bit quantity from host to network byte order
`ntohl(val)`	Convert 32-bit quantity from network to host byte order
`ntohs(val)`	Convert 16-bit quantity from network to host byte order

Byte-swapping routines are provided because ConvexOS expects addresses to be supplied in network order. (On some other architectures, host byte ordering is different from network byte ordering.) Consequently, programs are sometimes required to byte swap words. Library routines that return network addresses provide them in network order so that they may simply be copied into structures provided to the system. This implies users should encounter the byte-swapping problem only when interpreting network addresses. For example, to print an Internet port, the following code is required:

```
printf("port number%d\n", ntohs(sp->s_port));
```

Constructing client/server applications

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme, client applications request services from a server process. This section looks more closely at interactions between client and server, and considers some of the problems in developing client and server applications.

The client and server require a well-known set of conventions before service may be rendered or accepted. This set of conventions constitutes a protocol that is implemented at both ends of a connection. Depending on the situation, the protocol may be symmetrical or asymmetrical. In a symmetrical protocol, either side may play the master or slave role. An example of a symmetrical protocol is the `telnet` protocol used in the Internet domain for remote terminal emulation. In an asymmetrical protocol, one side is recognized as the master, with the other as the slave. An example of an asymmetrical protocol is the Internet file transfer protocol, `ftp`. No matter whether the specific protocol used in accessing a service is symmetrical or asymmetrical, there is always a client process and a server process. Server processes are described next, followed by a description of client processes.

Servers

This section describes the steps performed by the remote login server process shown in Figure 10.

A server process normally listens at a well-known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up" and serves the client, performing actions the client requests of it.

Alternative schemes using a service server may eliminate server processes that clog the system while remaining dormant most of the time. For Internet servers in ConvexOS, this scheme is implemented via `inetd`, the so called "Internet super-server." The daemon, `inetd`, listens at various ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which `inetd` is listening, `inetd` executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as `inetd` has played any part in the connection. `inetd` is described in more detail in Chapter 3, "Advanced socket programming," and in the inetd(8) man page.

Figure 10 Remote login server

```
main(argc, argv)

        int argc;
        char *argv[];
{
        int f;
        struct sockaddr_in from;
        struct servent *sp;

①       sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogind: tcp/login: unknown service\n");
                exit(1);
        }

         .
         .
         .
#ifndef DEBUG

        /* Disassociate server from controlling terminal */
②       for (i = 0; i < 3; ++i)
                close(i);

        open("/", O_RDONLY);
        dup2(0, 1);
        dup2(0, 2);

        i = open("/dev/tty", O_RDWR);
        if (i >= 0) {
                ioctl(i, TIOCNOTTY, 0);
                close(i);
        }

#endif
```

Figure 10 Remote login server (continued)

```
        sin.sin_port = sp->s_port;/* Restricted port--refer to Ch. 6 */
         .
         .
         .
③       f = socket(AF_INET, SOCK_STREAM, 0);
         .
         .
         .

        if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
④               .
                .
                .
        }
         .
         .
         .
⑤       listen(f, 5);

        for (;;) {
                int g, len = sizeof (from);

⑥              g = accept(f, (struct sockaddr *) &from, &len);
                if (g < 0) {
                        if (errno != EINTR)
                           syslog(LOG_DAEMON | LOG_ERR, "rlogind: accept: %m")
                        continue;
                }

⑦               if (fork() == 0) {
                        close(f);
                        doit(g, &from);
                }
                close(g);
        }
}
```

The following steps are identified in Figure 10:

① The server looks up its service definition by calling `getservbyname`. The result of the `getservbyname` call is later used to define the Internet port at which the server listens for service requests.

② The server disassociates from the controlling terminal of its invoker. This step is important because the server likely does not want to receive signals delivered to the process group of the controlling terminal. However, once a server has disassociated itself, it can no longer send reports of errors to a terminal, and must log errors via `syslog`.

③ Once a server has established a pristine environment, it creates a socket and begins accepting service requests.

④ The `bind` call is required to ensure that the server listens at its expected location. Within ConvexOS, most servers are accessed at well-known Internet addresses or UNIX domain names.

⑤ The remote login server listens at a restricted port number, and must therefore run as superuser. This concept of a "restricted port number" is discussed in Chapter 3, "Advanced socket programming."

⑥ An `accept` call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as `SIGCHLD` (discussed in Chapter 2, "Intermediate socket programming"). Therefore, the return value from `accept` is checked to ensure that a connection has actually been established. If an error occurs, it is logged via `syslog`.

⑦ With a connection in hand, the server forks a child process and invokes the main body of the remote login protocol processing. The socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the `accept` is closed in the parent. The address of the client is also passed to the `doit` routine, which authenticates clients.

Clients

This section describes the steps performed by the remote login client process shown in Figure 11. The separate, asymmetrical roles of client and server are clear in the code. The server is a passive entity, listening for client connections, whereas the client process is an active entity, initiating a connection when invoked.

Figure 11 Remote login client

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
  .
  .
  .
main(argc, argv)
        int argc;
        char *argv[];

{
        struct sockaddr_in server;
        struct servent *sp;
        struct hostent *hp;
        int s;
          .
          .
          .
①       sp = getservbyname("login", "tcp");
        if (sp == NULL) {
                fprintf(stderr, "rlogin: tcp/login: unknown service\n");
                exit(1);
        }

②       hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
                exit(2);
        }

③       bzero((char *)&server, sizeof (server));
        bcopy(hp->h_addr, (char *)&server.sin_addr, hp->h_length);
        server.sin_family = hp->h_addrtype;
        server.sin_port = sp->s_port;
```

Figure 11 Remote login client (continued)

```
④       s = socket(AF_INET, SOCK_STREAM, 0);
        if (s < 0) {
                perror("rlogin: socket");
                exit(3);
        }
          .
          .
          .
        /* Connect does the bind() for us */
⑤       if (connect(s, (struct sockaddr *)&server, sizeof (server)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
          .
          .
          .
}
```

① As in the server process, the first step is to locate the service definition for a remote login.

② The client looks up the destination host address by calling `getservbyname`.

③ The address buffer is cleared, then filled with the Internet address of the foreign host and the port number at which the login process resides on the foreign host.

④ The client creates a socket and initiates a connection.

⑤ `connect` implicitly performs a `bind` call, because `s` is unbound.

Intermediate socket programming

2

Introduction

Having considered basic aspects of socket programming in Chapter 1, "Basic socket programming," you are now ready to explore a slightly more difficult set of concepts that enable you to develop intermediate socket applications. In particular, this chapter explains in more detail the concepts of domain, communication style, and protocol. Understanding these concepts enables you to select programming options appropriate to your application.

The second part of the chapter explains in detail how to create, send, and receive datagrams in both the UNIX and Internet domains. (As Chapter 1 explains, datagrams are sockets that support a bidirectional flow of data that is not promised to be sequenced, reliable, or unduplicated.) This section contains numerous example programs that enable you to quickly learn the details of using datagrams.

The final part of this chapter explains how to develop connections for stream sockets, which provide bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Stream sockets must be connected to each other before data can be sent between them. Again, the discussion contains numerous programming examples.

Domains and protocols

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses. The space from which an address is drawn is referred to as a *domain*. ConvexOS supports two domains: the UNIX domain (or AF_UNIX, for Address Family UNIX) and the Internet domain (or AF_INET).

In the UNIX domain, a socket is given a path name within the file-system name space. A file-system node is created for the socket, and other processes may then refer to the socket by giving the proper path name. UNIX domain names, therefore, allow communication between any two processes that work in the same file system. The Internet domain is an implementation of the Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a *port*. Internet domain names allow communication between machines.

Communication follows a particular "style." Currently, communication is either through a stream or by datagram. Stream communication implies several things. Communication takes place across a connection between two sockets. The communication is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to `write`. If there is not enough room for the entire message, or if not all the data from a large message has been transferred, only part of the data from a single call will be read. The protocol implementing such a style retransmits messages received with errors. It also returns error messages if you try to send a message after the connection has been broken.

Datagram communication does not use connections; each message is addressed individually. If the address is correct, the message is generally received, although this is not guaranteed. Datagrams are often used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. Individual datagrams are kept separate when they are read; that is, message boundaries are preserved.

The difference in performance between the two communication styles is generally less important than the difference in semantics. Performance you might gain with datagrams must be weighed against the increased complexity of the program, which must concern itself with lost or out-of-order messages. If lost messages may be ignored, quantity of traffic may be a

consideration. The expense of setting up a connection is best justified by frequent use of the connection. Because performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

A *protocol* is a set of rules, data formats, and conventions that regulate the transfer of data between participants in the communication. There is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections, and transfers data between sockets, perhaps sending the data across a network.

Specify the domain and style of a socket when you create it. For example, in Figure 12, the call to `socket` creates a datagram socket in the UNIX domain.

Datagrams in the UNIX domain

Let us now look at two programs that create sockets. Programs in Figure 12 and Figure 13 use datagram, rather than stream, communication. The structure used to name UNIX domain sockets is defined in the <sys/un.h> file. The definition is also included in the example for clarity.

Each program creates a socket with a call to `socket`. These sockets are in the UNIX domain. Once a name has been decided upon, it is attached to a socket by the `bind` system call. The program in Figure 12 uses the name `socket`, which it binds to its socket. This name appears in the working directory of the program. The program in Figure 13 uses its socket only for sending messages. It does not create a name for the socket because no other process needs to refer to it.

Figure 12 Reading UNIX domain datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*
 * In the included file <sys/un.h> a sockaddr_un is defined as follows.
 * struct sockaddr_un {
 *      short    sun_family;
 *      char     sun_path[108];
 * };
 */

#include <stdio.h>
#define NAME "socket"
/*
 * This program creates a UNIX domain datagram socket, binds a name to it,
 * then reads from the socket.
 */

main()
{
        int sock, length;
        struct sockaddr_un name;
        char buf[BUFSIZ];
        /* Create socket from which to read. */
        sock = socket(AF_UNIX, SOCK_DGRAM, 0);
        if (sock < 0) {
                perror("opening datagram socket");
                exit(1);
        }
        /* Create name. */
        name.sun_family = AF_UNIX;
        strcpy(name.sun_path, NAME);
        if (bind(sock, &name, sizeof(struct sockaddr_un))) {
                perror("binding name to datagram socket");
                exit(1);
        }
        printf("socket -->%s\n", NAME);
        /* Read from the socket */
        if (read(sock, buf, BUFSIZ) < 0)
                perror("receiving datagram packet");
        printf("-->%s\n", buf);
        close(sock);
        unlink(NAME);
}
```

Figure 13 Sending a UNIX domain datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full..."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments. The form of the command line is udgramsend path name
 */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_un name;

        /* Create socket on which to send */
        sock = socket(AF_UNIX, SOCK_DGRAM, 0);
        if (sock < 0) {
                perror("opening datagram socket");
                exit(1);
        }

        /* Construct name of socket to send to. */
        name.sun_family = AF_UNIX;
        strcpy(name.sun_path, argv[1]);
        /* Send message. */
        if (sendto(sock, DATA, sizeof(DATA), 0, &name,
                           sizeof(struct sockaddr_un)) < 0) {
                perror("sending datagram message");
        }
        close(sock);
}
```

Names in the UNIX domain are path names. As with file path names, they may be either absolute (e.g., /dev/imaginary) or relative (e.g., `socket`). Because these names are used to allow processes to rendezvous, relative path names pose difficulties and should be used with care. When a name is bound into the name space, a file (`inode`) is allocated in the file system. If the `inode` is not deallocated, the name continues to exist even after the bound socket is closed. This can cause subsequent program runs to find that a name is unavailable, and can cause directories to fill with these objects. Names are removed by calling `unlink`

or using the `rm` command. Names in the UNIX domain are used only for rendezvous. They are not used for message delivery once a connection is established. Therefore, in contrast with the Internet domain, unbound sockets need not be (and are not) automatically given addresses when they are connected.

There is no established means of communicating names to interested parties. The program in Figure 13 gets the name of the socket to which it sends its message through its command-line arguments. Once a line of communication has been created, you can send names of additional, perhaps new, sockets over the link. Facilities must be built that make distribution of names less of a problem than it now is.

Datagrams in the Internet domain

Examples in Figure 14 and Figure 15 on the following pages are similar to previous examples except that the socket is in the Internet domain. The structure of Internet domain addresses is defined in the <netinet/in.h> file. Internet addresses specify a host address, as a 32-bit number, and a delivery slot, or port, on that machine. These ports are managed by system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed.

When a message must be sent between machines, it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but do not communicate with one another. As a result, different protocols may use the same port numbers. Thus, implicitly, an Internet address is an ordered set including port and machine address.

An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the ordered set <local machine address, local port, remote machine address, remote port>. An association may be transient when using datagram sockets; the association actually exists during a `send` operation.

Figure 14 Reading Internet domain datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
/*
 * In the included file <netinet/in.h> a sockaddr_in is defined as follows:
 * struct sockaddr_in {
 *      short    sin_family;
 *      u_short  sin_port;
 *      struct in_addr sin_addr;
 *      char     sin_zero[8];
 * };
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */

main()
{
        int sock, length;
        struct sockaddr_in name;
        char buf[1024];

        /* Create socket from which to read. */
        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
                perror("opening datagram socket");
                exit(1);
        }
        /* Create name with wildcards. */
        name.sin_family = AF_INET;
        name.sin_addr.s_addr = INADDR_ANY;
        name.sin_port = 0;
        if (bind(sock, &name, sizeof(name))) {
                perror("binding datagram socket");
                exit(1);
        }
        /* Find assigned port value and print it out. */
        length = sizeof(name);
        if (getsockname(sock, &name, &length)) {
                perror("getting socket name");
                exit(1);
        }
        printf("Socket has port #%d\n", ntohs(name.sin_port));
        /* Read from the socket */
        if (read(sock, buf, 1024) < 0)
                perror("receiving datagram packet");
        printf("-->%s\n", buf);
        close(sock);
}
```

Figure 15 Sending an Internet domain datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "The sea is calm tonight, the tide is full..."
/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments. The form of the command line is:
 *      dgramsend hostname portnumber
 */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_in name;
        struct hostent *hp, *gethostbyname();

        /* Create socket on which to send. */
        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
                perror("opening datagram socket");
                exit(1);
        }
        /*
         * Construct name, with no wildcards, of the socket to send to.
         * Getnostbyname returns a structure including the network address
         * of the specified host. The port number is taken from the command
         * line.
         */

        hp = gethostbyname(argv[1]);
        if (hp == 0) {
                fprintf(stderr, "%s: unknown host, argv[1]);
                exit(2);
        }

        bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
        name.sin_family = AF_INET;
        name.sin_port = htons(atoi(argv[2]));

        /* Send message. */
        if (sendto(sock, DATA, sizeof(DATA), 0, &name, sizeof(name)) < 0)
                perror("sending datagram message");
        close(sock);
}
```

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address for the machine, if it has more than one, or it can be the wildcard value, INADDR_ANY. The wildcard value is used in the program in Figure 14. If a machine has several network addresses, it is likely that messages sent to any of the addresses should be deliverable to a socket. This is the case if the wildcard value has been chosen. Even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address.

Programs can be willing to receive from "anywhere," but cannot send a message "anywhere." The program in Figure 15 is given the destination host name as a command-line argument. To determine a network address to which it can send the message, it looks up the host address by calling `gethostbyname`. The returned structure includes the host's network address, which is copied into the structure to specify the destination of the message.

The port number can be thought of as the street number on a mailbox, into which the protocol places messages. Daemons offering certain advertised services have reserved or "well-known" port numbers. These port numbers are from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system assigns an unused port number when an address is bound to a socket. This may happen when an explicit `bind` call is made with a port number of 0, or when a `connect` or `send` is performed on an unbound socket. Port numbers are not automatically reported back to the user. After calling `bind` asking for port 0, you may call `getsockname` to discover what port was actually assigned.

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of bytes in the address. Because machines differ in the internal representation of integers, printing the port number as returned by `getsockname` may result in a misinterpretation. To print the number, it is necessary to use the `ntohs` routine to convert the number from the network representation to the host's representation.

On CONVEX machines, this is a null operation. On other computers, such as VAXes, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format. It is called `htons`; similar routines exist for long integers. For further information, refer to the byteorder(3) man page.

Connections

To send data between stream sockets, sockets must be connected. Figure 16 and Figure 17 show two programs that create such a connection.

The program in Figure 16 is relatively simple. To initiate a connection, this program simply creates a stream socket, and then calls `connect`, specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, the connection is completed, and the program can begin to send messages. Messages are delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter).

If a process persists in sending messages after the connection is closed, the operating system sends a `SIGPIPE` signal to the process. Unless explicit action is taken to handle the signal, the process terminates and the shell prints the message, "`broken pipe`." Refer to the signal(3C) or sigvec(2) man pages for more information.

Figure 16 Initiating an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "Half a league, half a league..."
/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line. One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the command
 * line is:
 *      streamwrite hostname portnumber
 */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_in server;
        struct hostent *hp, *gethostbyname();
        char buf[1024];

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }

        /* Connect socket using name specified by command line. */
        server.sin_family = AF_INET;
        hp = gethostbyname(argv[1]);
        if (hp == 0) {
                fprintf(stderr, "%s: unknown host, argv[1]);
                exit(2);
        }

        bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
        server.sin_port = htons(atoi(argv[2]));
        if (connect(sock, &server, sizeof(server)) < 0) {
                perror("connecting stream socket");
                exit(1);
        }

        if (write(sock, DATA, sizeof(DATA)) < 0)
                perror("writing on stream socket");
        close(sock);
}
```

Figure 17 Accepting an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each time
 * through the loop it accepts a connection and prints messages from it.
 * When the connection breaks, or a termination message comes through, the
 * program accepts a new connection.
 */

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int rval;
        int i;

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }

        /* Name socket using wildcards */
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = 0;

        if (bind(sock, &server, sizeof(server))) {
                perror("binding stream socket");
                exit(1);
        }

        /* Find out assigned port number and print it out */
        length = sizeof(server);

        if (getsockname(sock, &server, &length)) {
                perror("getting socket name");
                exit(1);
        }
```

Figure 17 Accepting an Internet domain stream connection **(continued)**

```
        printf("Socket has port #%d\n", ntohs(server.sin_port));

        /* Start accepting connections */
        listen(sock, 5);

        do {
                msgsock = accept(sock, 0, 0);
                if (msgsock == -1)
                        perror("accept");
                else do {
                        bzero(buf, sizeof(buf));
                        if ((rval = read(msgsock, buf, 1024)) < 0)
                                perror("reading stream message");
                        i = 0;
                        if (rval == 0)
                                printf("Ending connection\n");
                        else
                                printf("-->%s\n", buf);
                } while (rval != 0);
                close(msgsock);
        } while (TRUE);
        /*
         * Since this program has an infinite loop, the socket "sock" is
         * never explicitly closed. However, all sockets are closed
         * automatically when a process is killed or terminates normally.
         */
}
```

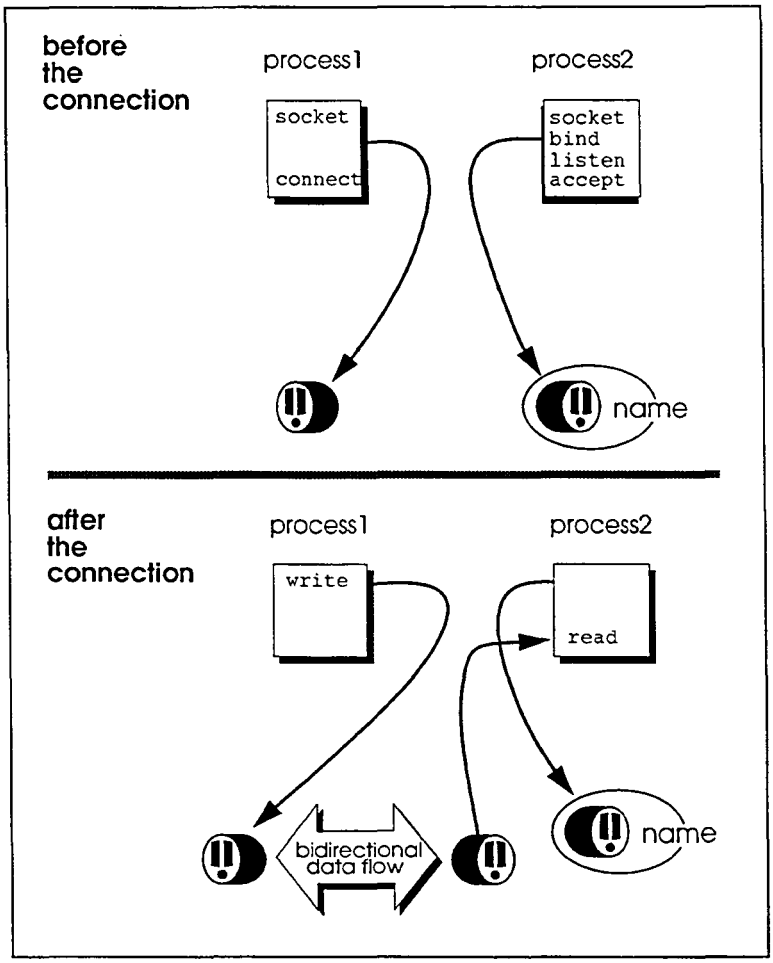
The program in Figure 17 is a simple example of a server. It creates a socket to which it binds a name, which it then advertises; in this case, it prints the socket number. The program then calls `listen` for this socket. Because several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. `listen` marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument passed to `listen`; the maximum length is limited by the system.

When the `listen` call has completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of Figure 18 shows the result of `process1` connecting with the named socket of `process2`, and `process2` accepting the connection.

After the connection is created, the service, in this case the one printing the messages, is performed and the connection socket closed. The `accept` call takes a pending connection request from the queue if one is available, or blocks waiting for a request. Messages are read from the connection socket. Read operations from an active connection normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the `read` call returns immediately, with the number of bytes set to zero.

Forming a connection is asymmetrical. One process, such as the program in Figure 16, requests a connection with a particular socket; the other process accepts connection requests. Before a connection can be accepted, a socket must be created and an address bound to it. This situation is illustrated in the top half of Figure 18.

Figure 18 Establishing a stream connection



In Figure 18, `process2` has created a socket and bound a port number to it. `process1` has created an unnamed socket. The address bound to `process2`'s socket is then made known to `process1` and perhaps to several other potential communicants as well. If there are several possible communicants, this one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection is destroyed by closing the corresponding socket.

The program in Figure 19 is a slight variation on the server in Figure 17. It avoids blocking when there are no pending connection requests by calling `select` to check for pending requests before calling `accept`. `select` takes five arguments: *nfds*, *readfds*, *writefds*, *exceptfds*, and *timeout*. *readfds*, *writefds*, and *exceptfds* are each `fd_sets`. An `fd_set` is a bitmap of file descriptor indexes. When you want to use `select` to learn when a socket is ready to have a connection accepted, use the `FD_SET` macro to set the bit corresponding to the socket's descriptor in the *readfds* `fd_set`. Do not set the bit in the *writefds* `fd_set` or the *exceptfds* `fd_set`. This strategy is useful when connections may be received on more than one socket, or when data may arrive on other connected sockets before another connection request.

Figure 19 Using `select` to check for pending connections

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program uses select to check that someone is trying to connect
 * before calling accept.
*/
main()

{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int rval;

        fd_set ready;
        struct   timeval to;

        /* Create socket */
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }
```

Figure 19 Using `select` to check for pending connections **(continued)**

```
        /* Name socket using wildcards */
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = 0;
        if (bind(sock, &server, sizeof(server))) {
                perror("binding stream socket");
                exit(1);
        }
        /* Find out assigned port number and print it */
        length = sizeof(server);
        if (getsockname(sock, &server, &length)) {
                perror("getting socket name");
                exit(1);
        }
        printf("Socket has port #%d\n", ntohs(server.sin_port));

        /* Start accepting connections */
        listen(sock, 5);
        do {
                FD_ZERO(&ready);
                FD_SET(sock, &ready);
                to.tv_sec = 5;
                if (select(sock + 1, &ready, 0, 0, &to) < 0) {
                        perror("select");
                        continue;
                }
                if (FD_ISSET(sock, &ready)) {
                      msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
                        if (msgsock == -1)
                                perror("accept");
                        else do {
                                bzero(buf, sizeof(buf));
                                if ((rval = read(msgsock, buf, 1024)) < 0)
                                     perror("reading stream message");
                                else if (rval == 0)
                                     printf("Ending connection\n");
                                else
                                     printf("-->%s\n", buf);
                        } while (rval > 0);
                        close(msgsock);
                } else
                        printf("Do something else\n");
        } while (TRUE);
}
```

Figure 20 and Figure 21 show a program using stream communication in the UNIX domain. Streams in the UNIX domain can be used for this sort of program in exactly the same way as Internet domain streams, except for the form of the names and the restriction of the connections to a single machine. There are some differences, however, in the functionality of streams in the two domains, notably that the UNIX domain does not support out-of-band data. (Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is discussed further in Chapter 3, "Advanced socket programming.")

Figure 20 Initiating a UNIX domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define DATA "Half a league, half a league..."

/*
 * This program connects to the socket named in the command line and sends a
 * one-line message to that socket. The form of the command line is:
 *
 *      ustreamwrite path name
 */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_un server;
        char buf[1024];

        /* Create socket */
        sock = socket(AF_UNIX, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }

        /* Connect socket using name specified by command line. */
        server.sun_family = AF_UNIX;
        strcpy(server.sun_path, argv[1]);

        if (connect(sock, &server, sizeof(struct sockaddr_un)) < 0) {
                close(sock);
                perror("connecting stream socket");
                exit(1);
        }

        if (write(sock, DATA, sizeof(DATA)) < 0)
                perror("writing on stream socket");
}
```

Figure 21 Accepting a UNIX domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a socket in the UNIX domain and binds a name to it.
 * After printing the socket's name, it begins a loop. Each time through the
 * loop it accepts a connection and prints messages from it. When the
 * connection breaks, or a termination message comes through, the program
 * accepts a new connection.
 */
main()
{
        int sock, msgsock, rval;
        struct sockaddr_un server;
        char buf[1024];

        /* Create socket */
        sock = socket(AF_UNIX, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("opening stream socket");
                exit(1);
        }

        /* Name socket using file system name */
        server.sun_family = AF_UNIX;
        strcpy(server.sun_path, NAME);
        if (bind(sock, &server, sizeof(struct sockaddr_un))) {
                perror("binding stream socket");
                exit(1);
        }
        printf("Socket has name %s\n", server.sun_path);
```

Figure 21 Accepting a UNIX domain stream connection **(continued)**

```
        /* Start accepting connections */
        listen(sock, 5);
        for (;;) {
                msgsock = accept(sock, 0, 0);
                if (msgsock == -1)
                        perror("accept");
                else do {
                        bzero(buf, sizeof(buf));
                        if ((rval = read(msgsock, buf, 1024)) < 0)
                                perror("reading stream message");
                        else if (rval == 0)
                                printf("Ending connection\n");
                        else
                                printf("-->%s\n", buf);
                } while (rval > 0);
                close(msgsock);
        }

        /*
         * The following statements are not executed, because they follow an
         * infinite loop. However, most ordinary programs will not run
         * forever. In the UNIX domain it is necessary to tell the file
         * system that one is through using NAME. In most programs one uses
         * the call unlink as below. Since the user must kill this
         * program, it is necessary to remove the name by a command from
         * the shell.
         */

        close(sock);
        unlink(NAME);
}
```

Advanced socket programming

3

Introduction

For many users, socket facilities discussed in Chapters 1 and 2 are sufficient for construction of required applications . Some users, however, need to consider some or all of the advanced topics discussed here:

- Addressing datagrams
- Using connectionless servers
- Using `select` to multiplex I/O requests
- Handling out-of-band data
- Using nonblocking sockets
- Using interrupt-driven socket I/O
- Using signals and process groups
- Using pseudoterminals
- Using `inetd`, the Internet "super-server" daemon
- Sending broadcast packets
- Using `setsockopt` and `getsockopt` to set and retrieve socket options
- Passing file descriptors

Datagram addressing

ConvexOS supports connectionless services typical of datagram facilities found in packet-switched networks. A datagram socket provides a symmetrical interface to data exchange. While processes are still likely to act as client and server, no connection need be established prior to communication. Instead, each message includes a destination address.

To use connectionless sockets, create datagrams just as you did with the connection-oriented model. If a particular local address is needed, the `bind` operation must precede the first data transmission. Otherwise, the system sets the local address and port when data is first sent. To send data, the `sendto` system call is used. The syntax of the `sendto` call is:

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to,
        &tolen);
```

s, *buf*, *buflen*, and *flags* parameters are used as before. *to* and *tolen* values indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors are reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance, when a network is unreachable), the call returns -1, and the global value `errno` contains an error number.

To receive messages on an unconnected datagram socket, the `recvfrom` system call is provided. Its syntax is:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from,
        &fromlen);
```

Once again, the *fromlen* parameter is handled in a value-result fashion, initially containing the size of the *from* buffer, and modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets may also use the `connect` call to associate a socket with a specific destination address. In this case, any data sent on the socket automatically addresses the connected peer. Only one connected address is permitted for each socket at one time; a second `connect` changes the destination address, and a `connect` to a null address (family `AF_UNSPEC`) disconnects. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end-to-end connection). `accept` and `listen` are not used with datagram sockets.

While a datagram socket is connected, errors from recent `send` calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with `getsockopt`, `SO_ERROR`, may be used to interrogate the error status. A `select` for reading or writing returns true when an error indication is received. The next operation returns the error, and the error status is cleared.

Connectionless servers

This section describes the use of a connectionless datagram-based broadcast using a server. The specific example used is the rwho service, which provides users with status information for hosts connected to a local area network. This service, while predicated on the ability to broadcast information to all hosts connected to a particular network, is of interest as an example of datagram sockets usage.

A user on any machine running the rwho server may find out the current status of a machine with the ruptime program. Typical output is shown in Figure 22.

Figure 22 Example ruptime output

```
rocky      up          10:10,      2 users,    load 0.27, 0.15, 0.14
boris      down        0:29
natasha    up          25+09:48,   4 users,    load 1.49, 1.43, 1.41
peabody    up          2+12:04,    8 users,    load 4.67, 5.13, 4.59
```

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The use of broadcast for such a task is inefficient, because all hosts must process each message, whether or not they use an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

A simplified form of the rwho server, rwhod, is shown in Figure 23. rwhod performs two separate tasks:

1. It acts as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to ensure they have been sent by another rwho server process. Packets are then time-stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, database interpretation routines assume the host is down and indicate such on status reports.
2. It supplies information regarding the status of its host. This involves periodically acquiring system status information, packaging it in a message, and broadcasting it on the local

network for other rwho servers to receive. The supply function is triggered by a timer and runs off a signal.

Figure 23 rwho server

```
main()
{
        .
        .
        .
    sp = getservbyname("who", "udp");
    np = getnetbyname("localnet");
    sin.sin_addr = inet_makeaddr(np->n_net, INADDR_ANY);
    sin.sin_port = sp->s_port;
        .
        .
        .
    s = socket(AF_INET, SOCK_DGRAM, 0);
        .
        .
        .
    on = 1;

    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) < 0) {
            syslog(LOG_DAEMON | LOG_ERR, "setsockopt SO_BROADCAST: %m");
            exit(1);
    }

    bind(s, (struct sockaddr *) &sin, sizeof (sin));
        .
        .
        .
    signal(SIGALRM, onalrm);
    onalrm();

    for (;;) {
            struct whod wd;
            int cc, whod, len = sizeof (from);
            cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0,
                  (struct sockaddr *)&from, &len);
            if (cc <= 0) {
                  if (cc < 0 && errno != EINTR)
                        syslog(LOG_DAEMON | LOG_ERR, " rwho d: recv: %m");
                  continue;
            }
```

Figure 23 rwho server **(continued)**

```
                if (from.sin_port != sp->s_port) {
                    syslog(LOG_DAEMON | LOG_ERR, " rwhod : %d: bad from port",
                                ntohs(from.sin_port));
                    continue;
                }
                  .
                  .
                  .

                if (!verify(wd.wd_hostname)) {
                    syslog(LOG_DAEMON | LOG_ERR, " rwhod:bad host name from %x",
                                ntohl(from.sin_addr.s_addr));
                    continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
                  .
                  .
                  .
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Locating system status information is somewhat involved, but uninteresting. Deciding where to transmit the resultant packet is somewhat problematical, however.

Status information must be broadcast on the local network. For networks that do not support broadcasting, another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate known neighbors based on status messages received from other rwho servers. Unfortunately, this requires some bootstrapping information, because a server has no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a network are freshly booted, no machine has any known neighbors and thus never receives, or sends, any status information.

This is the problem faced by the routing table management process in propagating routing status information. Unsatisfactory as it may be, the standard solution is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts that possibly are not

directly neighbors. If the server is able to support networks that provide a broadcast capability, as well as those that do not, then networks with an arbitrary topology may share status information. However, you must be concerned about loops. That is, if a host is connected to multiple networks, it receives status information from itself. This can lead to an endless, wasteful exchange of information.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This requires a separate copy of the server at each host and makes maintenance a severe headache. ConvexOS attempts to isolate host-specific information from applications by providing system calls that return the necessary information. An example of such a system call is `gethostname`, which returns the host's official name.

The `ioctl` call mechanism is used to find the collection of networks to which a host is directly connected. Furthermore, a local network broadcasting mechanism is implemented at the socket level. Combining these two features allows a process to broadcast in a site-independent manner on any directly-connected local network that supports broadcasting. This permits ConvexOS to solve the problem of deciding how to propagate status information in the case of `rwho`, or more generally in broadcasting. Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via appropriate `ioctl` calls. Specifics of such broadcasting are complex, however, and are covered subsequently.

Using `select` to multiplex I/O requests

Another facility for developing applications is the use of I/O requests multiplexed among multiple sockets and/or files. This is done using the `select` call, as shown in Figure 24.

Figure 24 Multiplexing I/O requests

```
#include <sys/time.h>
#include <sys/types.h>
        .
        .
        .
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
        .
        .
        .
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

`select` takes as arguments pointers to three descriptor sets:

- Descriptor set for files from which the caller wishes to read data
- Descriptor set for files to which the caller wishes to write data
- Descriptor set for exceptional conditions that may be pending. Out-of-band data is the only exceptional condition implemented for sockets. (Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Refer to "Out-of-band data" on page 55.)

If the user is not interested in certain conditions, for example, read, write, or exceptions, the corresponding argument to the `select` should be a null pointer.

Each set is actually a structure containing an array of long integer bit masks; the size of the array is set by the definition `FD_SETSIZE`. The array is long enough to hold one bit for each of `FD_SETSIZE` file descriptors.

Macros FD_SET(fd, &mask) and FD_CLR(fd, &mask) are provided for adding and removing file descriptor fd in the set mask. The set should be zeroed before use, and the macro FD_ZERO(&mask) is provided to clear the set mask. The parameter nfds in the select call specifies the range of file descriptors to be examined in a set, i.e., one plus the value of the largest descriptor.

Both the tv_sec and tv_usec fields of the timeval structure should be initialized prior to calling select.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in timeout are set to 0, the selection takes the form of a poll, returning immediately. If timeout is a null pointer, the selection blocks indefinitely; a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

select normally returns the number of file descriptors selected; if the select call returns due to the timeout expiring, then the value 0 is returned. If the select terminates because of an error or interruption, a -1 is returned with the error number in errno, and with the file descriptor masks unchanged.

Assuming a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask may be tested with the FD_ISSET(fd, &mask) macro, which returns a nonzero value if fd is a member of the set mask, and 0 if it is not.

To determine if connections are waiting on a socket to be used with an accept call, select can be used, followed by a FD_ISSET(fd, &mask) macro to check for read readiness on the appropriate socket. If FD_ISSET returns a nonzero value, indicating permission to read, then a connection is pending on the socket.

As an example, the code shown in Figure 25 might be used to read data from two sockets, s1 and s2, as it is available from each and with a one-second timeout.

Figure 25 Reading data from two sockets

```
#include <sys/time.h>
#include <sys/types.h>
        .
        .
        .
fd_set read_template;
struct timeval wait;
        .
        .
        .
for (;;) {
        wait.tv_sec = 1;   /* one second */
        wait.tv_usec = 0;

        FD_ZERO(&read_template);

        FD_SET(s1, &read_template);
        FD_SET(s2, &read_template);

        nb = select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
        if (nb <= 0) {
                /* An error occurred during the select, or
                the select timed out */

           .
           .
           .

        }
        if (FD_ISSET(s1, &read_template)) {
                /* Socket #1 is ready to be read from. */

           .
           .
           .

        }
        if (FD_ISSET(s2, &read_template)) {
                /* Socket #2 is ready to be read from. */
        }

           .
           .
           .

}
```

`select` provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of `SIGIO` and `SIGURG` signals described in the section "Signals and process groups" on page 60.

Out-of-band data

The stream socket abstraction includes the notion of out-of-band data for sockets in the Internet domain. Out-of-band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data is delivered to the user independently of normal data. The abstraction defines that out-of-band data facilities must support reliable delivery of at least one out-of-band message at a time. This message may contain one byte of data, and only one message may be pending delivery to the user at any one time.

For communication protocols that support only in-band signaling (i.e., urgent data is delivered in sequence with normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving urgent data in order and receiving it out of sequence without having to buffer all intervening data.

It is possible to "peek," via `MSG_PEEK`, at out-of-band data. If the socket has a process group, a `SIGURG` signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the `SIGURG` signal via the appropriate `fcntl` call, as described below for `SIGIO`. If multiple sockets may have out-of-band data awaiting delivery, a `select` call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the `select` indicates the actual arrival of out-of-band data, but only notification that it is pending.

In addition to information passed, a logical mark is placed in the data stream to indicate the point at which out-of-band data was sent. The remote login application uses this facility to propagate signals between client and server processes. When a signal flushes any pending output from the remote process or processes, all data up to the mark in the data stream is discarded.

To send an out-of-band message, the `MSG_OOB` option is supplied to a `send` or `sendto` call, while to receive out-of-band data, `MSG_OOB` should be indicated when performing a `recvfrom` or `recv` call. To find out if the read pointer is currently pointing at the mark in the data stream, the `SIOCATMARK ioctl` is provided.

```
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is a 1 on return, the next `read` returns data after the mark. Otherwise (assuming out-of-band data has arrived), the next `read` provides data sent by the client prior to transmission of the out-of-band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 26. It reads normal data up to the mark (to discard it), and then reads the out-of-band byte.

Figure 26 Flushing terminal I/O on receipt of out-of-band data

```
#include <sys/ioctl.h>
#include <sys/file.h>
        .
        .
        .
oob()
{
        int out = FWRITE;
        char waste[BUFSIZ], mark;

        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof (waste));
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                perror("recv");
                  .
                  .
                  .
        }
           .
           .
           .
}
```

A process may also read or peek at out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers urgent data in-band with normal data, and only sends notification of its presence ahead of time, as with the TCP protocol used to implement stream sockets in the Internet domain. With such protocols, the out-of-band byte may

not yet have arrived when a `recv` is done with the `MSG_OOB` flag. In that case, the call returns an error of `EWOULDBLOCK`. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., `telnet`) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, `SO_OOBINLINE`; refer to the setsockopt(2) man page for usage. With this option, the position of the urgent data (the mark) is retained, but the urgent data immediately follows the mark within the normal data stream returned without the `MSG_OOB` flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data is lost.

Nonblocking sockets

It is occasionally convenient to make use of sockets that do not block; that is, I/O requests that cannot complete immediately and that would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the `socket` call, it may be marked as nonblocking by using `fcntl` as shown in Figure 27.

Figure 27 Performing nonblocking I/O on sockets

```
#include <fcntl.h>
          .
          .
          .
int     s;
          .
          .
          .
s = socket(AF_INET, SOCK_STREAM, 0);
          .
          .
          .
if (fcntl(s, F_SETFL, FNDELAY) < 0)
        perror("fcntl F_SETFL, FNDELAY");
        exit(1);
}
```

When performing nonblocking I/O on sockets, be careful to check for the error `EWOULDBLOCK` (stored in the global variable `errno`), which occurs when an operation would normally block, but the socket it was performed on is marked as nonblocking. In particular, `accept`, `connect`, `send`, `recv`, `read`, and `write` can all return `EWOULDBLOCK`, and processes should be prepared to deal with such return codes. If an operation such as a `send` cannot be done in its entirety, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately is processed, and the return value indicates the amount actually sent.

Interrupt-driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Using the SIGIO facility requires three steps:

1. The process must set up a SIGIO signal handler by use of the `signal` or `sigvec` calls.
2. It must set the process id or process group id that is to receive notification of pending input to its own process id, or the process group id of its process group (the default process group of a socket is group zero). This is accomplished by use of an `fcntl` call.
3. It must enable asynchronous notification of pending I/O requests with another `fcntl` call.

Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket `s` is shown in Figure 28. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

Figure 28 Using asynchronous notification of I/O requests

```
#include <fcntl.h>
          .
          .
          .

int     io_handler();
          .
          .
          .

signal(SIGIO, io_handler);

/* Set the process receiving SIGIO/SIGURG signals to us */

if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}

/* Allow receipt of asynchronous I/O signals */

if (fcntl(s, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
}
```

Signals and process groups

Because of the existence of `SIGURG` and `SIGIO` signals, each socket has an associated process number (as do terminals). This value is initialized to zero, but may be redefined at a later time with the `F_SETOWN fcntl`, as was done in the code above for `SIGIO`. To set the socket's process id for signals, give positive arguments to the `fcntl` call. To set the socket's process group for signals, negative arguments should be passed to `fcntl`. The process number indicates either the associated process id or the associated process group; it is impossible to specify both at the same time. A similar `fcntl`, `F_GETOWN`, is available for determining the current process or process group number of a socket.

Another useful signal when constructing server processes is `SIGCHLD`. This signal is delivered to a process when any child process has changed state. Normally servers use the signal to reap child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Chapter 1, "Introductory Socket Programming," may be augmented as shown in Figure 29.

Figure 29 Using the SIGCHLD signal

```
int reaper();
        .
        .
        .
signal(SIGCHLD, reaper);
listen(f, 5);
for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len,);
        if (g < 0) {
                if (errno != EINTR)
                        syslog(LOG_DAEMON | LOG_ERR, "rlogind: accept: %m");
                continue;
        }
           .
           .
           .
}
          .
          .
          .
#include <wait.h>
reaper()
{
        union wait status;

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

Be aware that if the parent server process fails to reap its child processes, a large number of zombie processes may be created.

Pseudoterminals

Many programs do not function properly without a terminal for standard input and output. Because sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a pseudoterminal.

A *pseudoterminal* is actually a pair of devices, master and slave, that allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudoterminal is supplied as input to a process reading from the master side, whereas data written on the master side is processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudoterminal controls information read and written on the slave side as if it were manipulating the keyboard and reading the screen on an actual terminal. This abstraction preserves terminal semantics over a network connection—that is, the slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudoterminals for remote login sessions. A user logging in to a machine across the network is provided a shell with a slave pseudoterminal as standard input, output, and error. The server process then handles communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that generates an interrupt and flushes terminal output on the remote machine, the pseudoterminal generates a control message for the server process. The server then sends an out-of-band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under ConvexOS, the name of the slave side of a pseudoterminal is of the form `/dev/ttyxy`, where *x* is a single letter from the set "pqrstonmlkjihgfe," and *y* is a hexadecimal digit, i.e., a single character in the range 0 through 9 or "a" through "f." The master side of a pseudoterminal is `/dev/ptyxy`, where *x* and *y* correspond to the slave side of the pseudoterminal.

In general, the method of obtaining a pair of pseudoterminals is to find an unused pseudoterminal. The master half of a pseudoterminal is a single-open device; thus, you could open each master in turn until an open succeeds, or you could use `getpty` as described below. Once you have found an unused pseudoterminal for the master side of the pair, the slave side of the pseudoterminal is then opened, and set to the proper

terminal modes if necessary. The process then forks; the child closes the master side of the pseudoterminal, and execs the appropriate program. Meanwhile, the parent closes the slave side of the pseudoterminal and begins reading and writing from the master side.

Using `getpty` is preferable to using successive `open` calls because `getpty` functions without regard to the number of pseudoterminals configured into your system, and independently of `pty` naming conventions described above. Be aware when using `getpty` that the name it returns is advisory only; it does not guarantee that a call to `open` will succeed.

Figure 30 shows sample code making use of pseudoterminals; this code assumes that a connection on socket `s` exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

Figure 30 Creating and using a pseudoterminal

```
/*
 * open master side.
 */
for (;;) {
        if ((line = getpty()) == NULL) {
                syslog(LOG_DAEMON | LOG_ERR, "All network ports in use");
                exit(1);
        }
        if ((master_fd = open (line, 2)) >= 0) {
                /*
                 * open slave side.
                 */
                line[5] = 't';      /* /dev/ptyXY ==> /dev/ttyXY */
                if ((slave_fd = open (line, 2)) >= 0)
                        break;
                else
                        close (master_fd);
        }
}
line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR);          /* slave is now slave side */
if (slave < 0) {
        syslog(LOG_DAEMON | LOG_ERR, "Cannot open slave pty %s", line);
        exit(1);
}
ioctl(slave, TIOCGETP, &b);          /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);
i = fork();
if (i < 0) {
        syslog(LOG_DAEMON | LOG_ERR, "fork: %m");
        exit(1);
} else if (i) {                      /* Parent */
        close(slave);
           .
           .
           .
} else {                             /* Child */
        (void) close(s);
        (void) close(master);
        dup2(slave, 0);
        dup2(slave, 1);
        dup2(slave, 2);
        if (slave > 2)
                (void) close(slave);
           .
           .
           .
}
```

Using inetd

One of the daemons provided with ConvexOS is inetd, the so-called "Internet super-server." Invoked at boot time, inetd determines from the /etc/inetd.conf file the servers for which it is to listen. Once this information has been read and a pristine environment created, inetd creates one socket for each service it is to listen for, binding the appropriate port number to each socket. inetd is useful because it eliminates system overhead associated with multiple server processes waiting to accept connections. The format of the /etc/inetd.conf file is described in the inetd(8) man page.

inetd next performs a select on all its sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. inetd then performs an accept on the socket in question, forks, dups the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and execs the appropriate server.

Servers making use of inetd are considerably simplified, because inetd takes care of the majority of the IPC work required in establishing a connection. The server invoked by inetd expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as read, write, send, or recv. Indeed, servers may use buffered I/O as provided by stdio conventions, as long as they remember to use fflush when appropriate.

One call that may be of interest when writing servers under inetd is the getpeername call, which returns the address of the peer process connected on the other end of the socket. For example, to log the Internet address in dot notation (e.g., 128.32.0.4) of a client connected to a server under inetd, the code shown in Figure 24 might be used.

Figure 31 Obtaining the address of a peer process

```
struct sockaddr_in name;
int namelen = sizeof (name);
          .
          .
          .
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
        syslog(LOG_DAEMON | LOG_ERR, "getpeername: %m");
        exit(1);
} else
        syslog(LOG_DAEMON | LOG_INFO, "Connection from %s",
                inet_ntoa(name.sin_addr));
          .
          .
          .
```

While the `getpeername` call is especially useful when writing programs to run with `inetd`, it can be used under other circumstances.

Broadcasting and determining network configuration

By using a datagram socket, it is possible to send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network because they force every host on the network to service them. Consequently, the ability to send broadcast packets is limited to sockets that are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons: you need to find a resource on a local network without prior knowledge of its address, or important functions such as routing require that information be sent to all accessible neighbors.

To send a broadcast message, create a datagram socket, as follows:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int  on = 1;

setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

and at least a port number should be bound to the socket.

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The destination address of the broadcast message depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address `INADDR_BROADCAST`, defined in the <netinet/in.h> file. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Because this information should be obtained in a host-independent fashion and may be impossible to derive, ConvexOS provides a method of retrieving this information from system data structures. The `SIOCGIFCONF ioctl` call returns the interface configuration of a host in the form of a single `ifconf` structure; this structure contains a "data area" made up of an array of `ifreq` structures, one for each network interface to which the host is connected. These structures are defined in the <net/if.h> file as shown in Figure 32.

Figure 32 The <net/if.h> file

```
struct ifconf {
        int     ifc_len;                        /*size of associated buffer*/
        union {
                caddr_t ifcu_buf;
                struct ifreq *ifcu_req;
        } ifc_ifcu;
};

#define ifc_buf ifc_ifcu.ifcu_buf               /*buffer address*/
#define ifc_req
ifc_ifcu.ifcu_req                               /* array of structs returned */

#define IFNAMSIZ  16

struct ifreq {
        char    ifr_name[IFNAMSIZ];             /* if name, e.g. "en0" */
        union {
                struct sockaddr ifru_addr;
                struct sockaddr ifru_dstaddr;
                struct sockaddr ifru_broadaddr;
                short ifru_flags;
                int ifru_metric;
                caddr_t ifru_data;
                struct if_subnet ifru_subnet;
        } ifr_ifru;
};

#define ifr_addr ifr_ifru.ifru_addr             /* address */
#define ifr_dstaddr ifr_ifru.ifru_dstaddr       /* other end of p-to-p link */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr   /* broadcast address */
#define ifr_flags ifr_ifru.ifru_flags           /* flags */
#define ifr_metric ifr_ifru.ifru_metric         /* metric */
#define ifr_data ifr_ifru.ifru_data             /* for use by interface */
#define ifr_subnet ifr_ifru.ifru_subnet.subnet
#define ifr_subnetmask ifr_ifru.ifru_subnet.subnetmask
```

The actual call that obtains the interface configuration is shown in Figure 33.

Figure 33 Determining network interface configuration

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
        .
        .
        .
}
```

After this call, `buf` contains one `ifreq` structure for each network to which the host is connected, and `ifc.ifc_len` has been modified to reflect the number of bytes used by the `ifreq` structures.

For each structure, a set of "interface flags" tells whether the network corresponding to that interface is up or down, point-to-point or broadcast, etc. The `SIOCGIFFLAGS ioctl` retrieves these flags for an interface specified by an `ifreq` structure as shown in Figure 34.

Figure 34 Obtaining interface status flags

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
        if (ifr->ifr_addr.sa_family != AF_INET)
                continue;
        if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
                .
                .
                .
        }
        /*
         * Skip boring cases.
         */
        if ((ifr->ifr_flags & IFF_UP) == 0 ||
                (ifr->ifr_flags & IFF_LOOPBACK) ||
                (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
                        continue;
```

After the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks, this is done via the `SIOCGIFBRDADDR ioctl`, whereas for point-to-point networks, the address of the destination host is obtained with `SIOCGIFDSTADDR`, as shown in Figure 35.

Figure 35 Obtaining broadcast or destination address

```
struct sockaddr dst;

if (ifr->ifr_flags & IFF_POINTTOPOINT) {
        if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
                .
                .
                .
        }
        bcopy((char *) ifr->ifr_dstaddr, (char *) &dst,
                  sizeof (ifr->ifr_dstaddr));
} else if (ifr->ifr_flags & IFF_BROADCAST) {
        if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
                .
                .
                .
        }
        bcopy((char *) ifr->ifr_broadaddr, (char *) &dst,
                  sizeof (ifr->ifr_broadaddr));
}
```

After appropriate `ioctl` calls have obtained the broadcast or destination address (now in `dst`), the `sendto` call may be used, as follows:

```
sendto(s, buf, buflen, 0,
       (struct sockaddr *)&dst, sizeof (dst));
```

One `sendto` call occurs for every interface to which the host is connected that supports the notion of broadcast or point-to-point addressing. If a process wished only to send broadcast messages on a given network, code similar to that outlined above is used, but the loop needs to find the correct destination address.

Received broadcast messages contain the sender's address and port, because datagram sockets are bound before a message is allowed to go out.

Socket options

It is possible to set and get a number of options on sockets via the `setsockopt` and `getsockopt` system calls. These options, defined in </sys/sockets.h>, include such things as whether or not to mark a socket for broadcasting, whether or not to route, whether or not to linger on close, etc. Table 3 lists options supported by Internet Services protocols.

Table 3 Options supported by Internet Services protocols

Option level	Option name	Used by TCP	Used by UDP
INET_GENERIC	SO_BROADCAST	x	x
	SO_DEBUG	x	x
	SO_DONTROUTE	x	x
	SO_LINGER	x	
	SO_KEEPALIVE	x	
	SO_RCVBUF	x	x
	SO_SNDBUF	x	x
	SO_SNDLOWAT	x	x
	SO_RCVLOWAT	x	x
	SO_REUSEADDR	x	x
	SO_USELOOPBACK	x	x
INET_TCP	TCP_NODELAY	x	
	TCP_MAXSEG	x	
INET_UDP	UDPCHECKSUM		x
INET_IP	IP_TOS	x	x
	IP_TTL	x	x
	IP_OPTIONS	x	x

The general forms of the calls for socket options are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

where:

s	Socket on which to apply the option.
level	Protocol layer on which to apply the option; in most cases, this is the "socket level," indicated by the symbolic constant `SOL_SOCKET`, defined in the <sys/socket.h> file.
optname	Actual option in the form of a symbolic constant also defined in the <sys/socket.h> file.

For `getsockopt`, *optval* and *optlen* point to the value of the option (in most cases, whether the option is to be turned on or off), and the length of the value of the option, respectively. For `getsockopt`, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified on return to indicate the actual amount of storage used.

For `setsockopt`, neither *optlen* nor *optval* is a pointer. *optlen* is the length of the value of the option, not a pointer to that length. *optval* points to a buffer containing the option value.

An example helps clarify things. It is sometimes useful to determine the type, e.g., stream, datagram, etc., of an existing socket; programs under control of `inetd` (described below) may need to perform this task. This can be accomplished as follows via the `SO_TYPE` socket option and the `getsockopt` call, as shown in Figure 36.

Figure 36 Determining socket type

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;
size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0){
        .
        .
        .
}
```

After the `getsockopt` call, `type` is set to the value of the socket type, as defined in the <sys/socket.h> file. For example, if the socket were a datagram socket, `type` would have the value corresponding to `SOCK_DGRAM`.

Options available for a socket depend upon its communication domain and socket type. The UNIX domain supports `SO_SNDBUF`, `SO_RCVBUF`, `SO_ERROR`, and `SO_TYPE` options. In the Internet domain, all socket options are supported when you are using a stream socket; datagram sockets support only `SO_DONTROUTE`, `SO_BROADCAST`, `SO_SNDBUF`, `SO_RCVBUF`, `SO_REUSEADDR`, `SO_ERROR`, and `SO_TYPE` options.

The `SO_OOBINLINE` option is only supported for stream sockets in the Internet domain. When it is enabled, the socket can only receive out-of-band data as regular data; however, both the select for exceptional conditions and the `SIGURG` signal still apply. Out-of-band data is returned as one octet.

A socket's type and communication domain also influence actions taken when the socket receives a message larger than its receive buffer (as specified with the `SO_RCVBUF` option). Because stream data is reassembled at the socket layer, a stream socket could receive messages larger than its receive buffer. In the UNIX domain, if a socket receives a datagram larger than its receive buffer, an `ENOBUFS` error will occur. However, in the Internet domain, receiving a datagram larger than a socket's receive buffer will not result in an error, but the socket will not receive the datagram.

The `SO_REUSEADDR` option allows you to bind a socket to an address bound to another socket that has been closed, but its process control block still exists. This may be necessary because sometimes a small window exists between closing a socket and the time the address bound to it is released.

When the SO_KEEPALIVE option is enabled, the underlying protocol transmits periodic messages at 45-second intervals. If no response is received after transmission of 8 such messages, the connection is considered broken, and processes using the socket are notified via the SIGPIPE signal.

Passing file descriptors

Within the UNIX domain, you can use sockets to pass access rights. Access rights refer to file descriptors a process "owns" or has access to. These ownership rights can be passed via sockets (using sendmsg and recvmsg) between processes. In this way, the ability to access needed files can be shared between processes.

The ability to pass file descriptors allows processes to access files they would otherwise be restricted from accessing. A client program can contact a server and request open descriptors for files to which the server, and not the client, has access.

Figure 37 and Figure 38 illustrate how access rights can be transferred between client and server.

Figure 37 `client.c`: transmitting access rights

```
/*
 * acc_client -- opens a file, then transmits its descriptor
 * through the access rights of a datagram type
 * unix domain socket.
 */
#include <sys/types.h>
#include <sys/file.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/uio.h>
#define SOCKET_NAME "Delilah"

main(argc, argv)
int argc;
char *argv[];
{
        int i, fd, s;
        struct msghdr msg;
        struct sockaddr_un s_un;
        struct iovec iov;
        char buf[1], strcpy();
        if ((s = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
                perror("socket");
                exit(1);
        }

        s_un.sun_family = AF_UNIX;
        (void) strcpy(s_un.sun_path, SOCKET_NAME);
        for (i = 1; i < argc; i++)
                if ((fd = open(argv[i], O_RDONLY, 0)) < 0)
                        perror(argv[i]);
                else {
                        msg.msg_accrights = (caddr_t) &fd;
                        msg.msg_accrightslen = sizeof fd;
                        msg.msg_name = (caddr_t) &s_un;
                        msg.msg_namelen = sizeof s_un;
                        iov.iov_base = buf;
                        iov.iov_len = 0;
                        msg.msg_iov = &iov;
                        msg.msg_iovlen = 1;
                        if (sendmsg(s, &msg, 0) < 0) {
                                perror("sendmsg");
                                exit(1);
                        }
                        (void) close(fd);
                }
        return 0;
}
```

Figure 38 `acc.server`: receiving access rights

```
/*
 * acc_server -- receives file descriptors in the access rights
 * of unix domain datagrams, then copies from the
 * file descriptors to standard output.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/uio.h>

#define SOCKET_NAME "Delilah"

main()
{
        int newfd, s, n;
        struct sockaddr_un s_un;
        struct msghdr msg;
        char buf[1024];
        struct iovec iov;
        extern char *strcpy();

        if ((s = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
                perror("socket");
                exit(1);
        }

        s_un.sun_family = AF_UNIX;
        (void) strcpy(s_un.sun_path, SOCKET_NAME);
        (void) unlink(SOCKET_NAME);

        if (bind(s, (struct sockaddr *) &s_un, sizeof s_un)) {
                perror("bind");
                exit(1);
        }

        while (1) {
                msg.msg_accrights = (caddr_t) &newfd;
                msg.msg_accrightslen = sizeof newfd;
                msg.msg_name = 0;
                msg.msg_namelen = 0;
                iov.iov_base = buf;
                iov.iov_len = sizeof buf;
                msg.msg_iov = &iov;
                msg.msg_iovlen = 1;
```

Figure 38 `acc.server`: receiving access rights **(continued)**

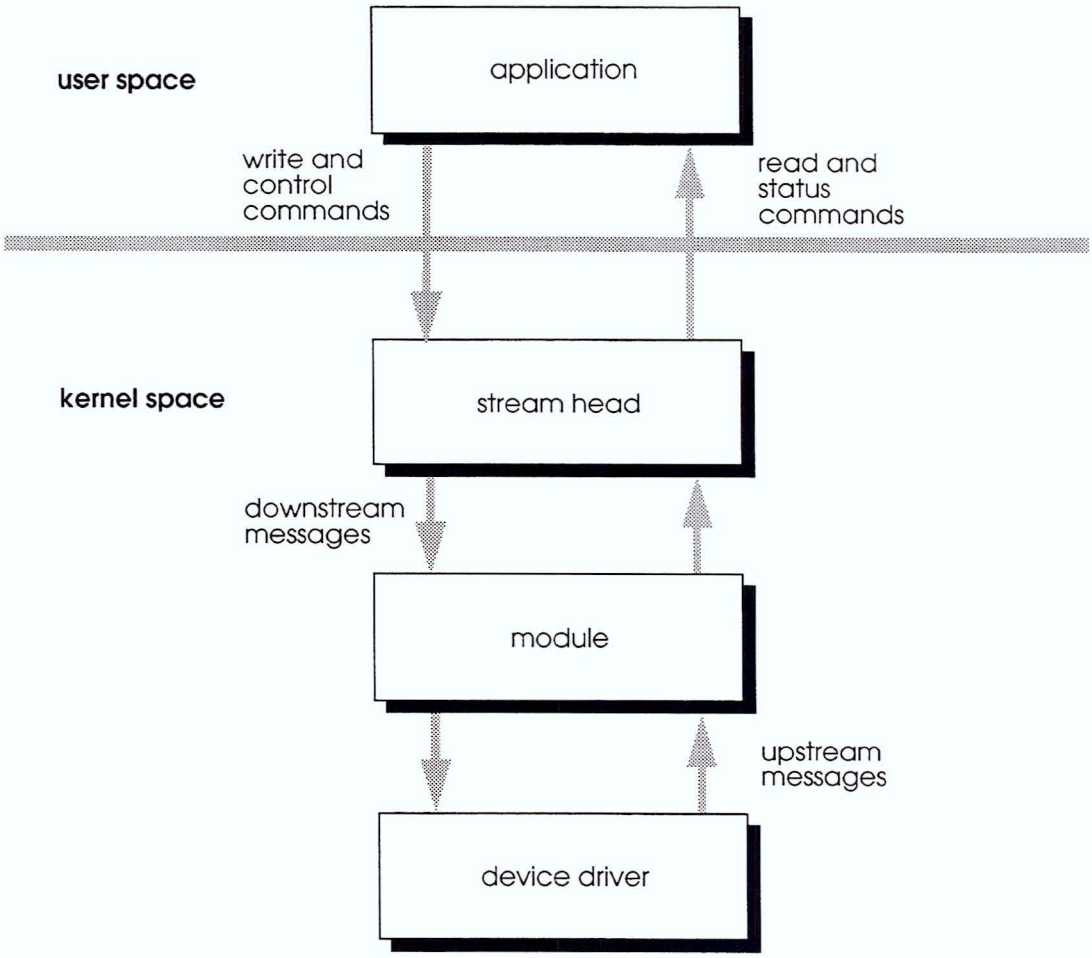
```
                if (recvmsg(s, &msg, 0) < 0) {
                        perror("recvmsg");
                        exit(1);
                }

                while ((n = read(newfd, buf, sizeof buf)) > 0)
                        (void) write(1, buf, n);

                if (n < 0) {
                        perror("read");
                        exit(1);
                }

                if (close(newfd)) {
                        perror("close");
                        exit(1);
                }
        }
}
```

STREAMS interface



STREAMS interface

STREAMS is a kernel mechanism that provides a highly modular and flexible framework for network services and data communications. STREAMS also provides an enhanced system call interface through which user-level applications access these services.

This part explains how to use the ConvexOS STREAMS interface for network programming applications:

- Chapter 4 introduces basic STREAMS terms and concepts.
- Chapter 5 explains using the STREAMS system call interface for programming STREAMS applications
- Chapter 6 explains how to develop STREAMS applications for TCP/IP using the Network Provider and Transport Provider Interfaces.
- Chapter 7 explains how to convert raw socket applications to STREAMS.

Introduction to STREAMS

4

Overview

STREAMS consists of system calls, kernel resources, and kernel routines. It defines standard interfaces for character I/O within the kernel, and between the kernel and user space.

STREAMS works well for implementing multilayer network protocols because of its modular structure. Each layer of a protocol stack can be defined as a *module*, or processing component, of a stream. A *stream* refers to the path between a user application and a driver, while STREAMS refers to the kernel facility.

Modules are connected and operate on messages passed between the user and the STREAMS driver. The sequence of modules on a stream can be dynamically configured.

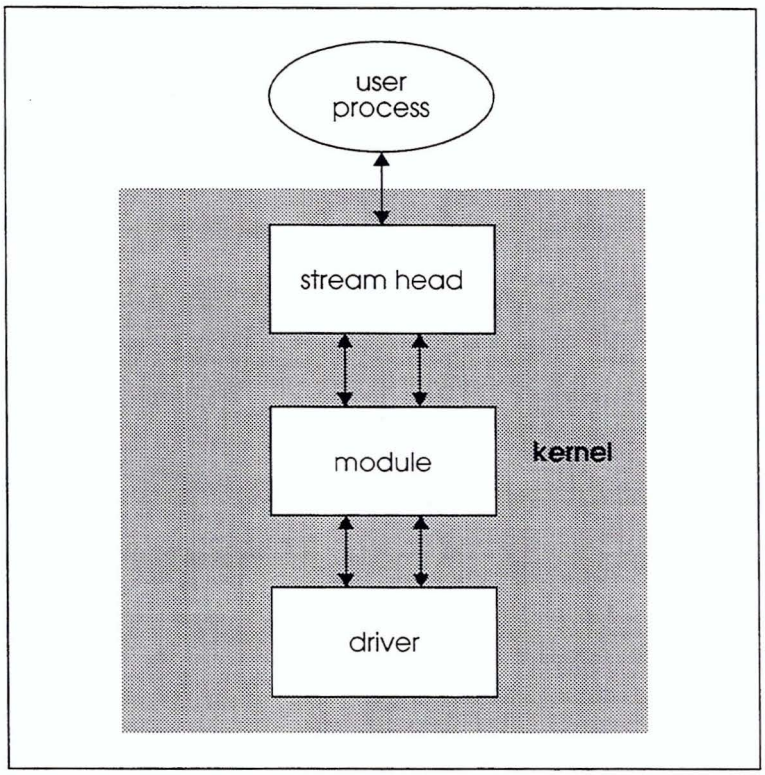
STREAMS components

A *stream* is a full-duplex processing and data communications path between a user process and a driver in the kernel. A stream has three parts:

- Stream head
- One or more optional modules
- Driver

Figure 39 shows a basic stream.

Figure 39 Basic stream



The *stream head* is the interface between the stream in kernel space and a user process. It processes system calls made by a user-level process on a stream and performs bidirectional transfer of data and information between the stream and the user process.

Modules, consisting of data structures and kernel routines, are the processing components in STREAMS. They are optional (if there is no module, the driver performs all character and device processing). Modules can process data by changing the way it is represented, adding or removing header and trailer information, and assembling or disassembling packets. Modules also process status and control information, including signals and input/output control information.

A STREAMS *driver* can be a device driver, associated with a hardware device such as an Ethernet controller, or an internal software driver, also known as a pseudo-device driver. The device driver transfers data between the kernel and the device, converting data structures used by the device into STREAMS data structures. Pseudo-device drivers are usually multiplexing drivers (described later in this section) and do not interface directly to hardware. STREAMS drivers are opened in the same manner as conventional device drivers.

In the CONVEX implementation of the TCP/IP protocol stack, Ethernet is a device driver, and TCP and IP are pseudo-device drivers.

Queues and messages

Each of the three STREAMS components—the stream head, modules, and driver—maintains an adjacent pair of queues to keep information about STREAMS operations. A queue is a kernel data structure that stores messages for processing. The queue pair is allocated when a driver is opened or a module is pushed onto a stream. One queue contains messages moving from the user to the device driver; it is referred to as the *downstream* or write queue. The other queue contains messages moving from the device driver to the user; it is referred to as the *upstream* or read queue. STREAMS system calls operate directly on these queues and the messages they contain.

STREAMS components communicate by passing messages containing data and control information between their queues. The order in which messages are placed on a queue is determined by the priority of the message. Messages are classified as priority or ordinary. Priority messages are placed at the front of the queue after all other priority messages; ordinary messages are always placed at the back of the queue.

STREAMS stacks

STREAMS enables a user process to add or remove in last-in-first-out (LIFO) order one or more modules between the stream head and the driver to perform intermediate processing on data flowing in the stream. Modules are inserted and deleted at the stream head with the `ioctl` system call; in ConvexOS, the `knetdctl` utility takes care of issuing the `ioctl` calls to link modules and drivers. The information that the `knetdctl` utility uses to build the STREAMS stack is specified in the configuration file /etc/knetd.conf.

For more information on constructing STREAMS stacks, refer to *Managing Internet Services and NFS*.

Multiplexors

When a stream is initialized by opening a STREAMS device, by default all subsequent `open` calls to that device return a file descriptor that references the same stream. Each process that opens the same minor device shares the same stream to the device driver. Some applications, such as configuring multi-layer protocols, require the ability to send and receive data from multiple streams. A STREAMS device that services multiple streams is called a *multiplexor*.

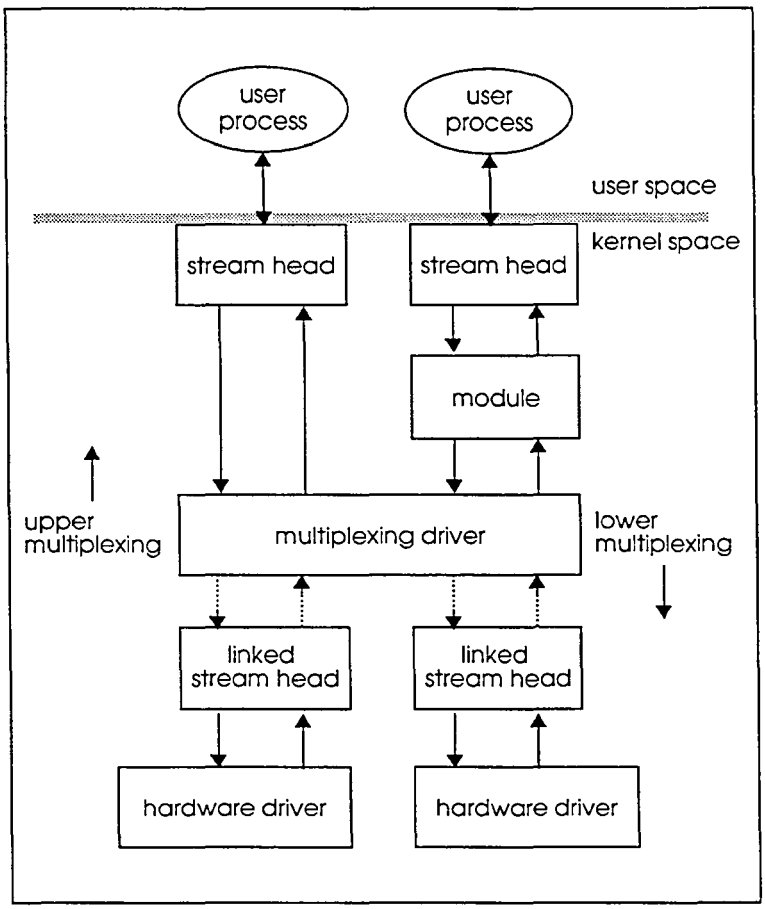
A multiplexor that services multiple streams between the user and the multiplexor and sends messages down a single stream to the driver is referred to as an *upper multiplexor* or *N-to-1* multiplexor. The CONVEX TCP driver is an upper multiplexor.

A multiplexor that services multiple streams between the multiplexor and the device hardware is referred to as a *lower multiplexor* or *1-to-N* multiplexor. Lower multiplexors are constructed and dismantled using `I_LINK` and `I_UNLINK` `ioctl` calls. As explained in the previous section, the `knetdctl` utility links streams to the multiplexor from information in a configuration file.

A third type of multiplexor can route data from one of many upper streams to one of many lower streams. For example, in the CONVEX TCP/IP stack, several streams from the Ethernet driver are linked under the IP driver, and the IP driver is linked below UDP and TCP.

Figure 40 illustrates a multiplexor capable of upper and lower multiplexing.

Figure 40 Upper and lower multiplexing



Clonable devices

Like a character I/O driver, a STREAMS driver is associated with one or more nodes in the file system, with each node corresponding to a separate minor device for that driver. For each minor device of a driver that is opened, a separate stream is connected between a user process and the driver. Subsequent open calls to the same minor device return a file descriptor that references the stream that was created when the device was first opened. All processes that open the same minor device share the same stream to the driver.

There are times, however, when an application may need to open a separate stream to the driver on every `open` call. This is accomplished by opening an unused minor device of the driver.

To avoid specifing the minor device by name in the `open` call, STREAMS drivers can be implemented as *clonable* devices. Opening a clonable device results in a call to the STREAMS driver open routine with the `CLONEOPEN` option. The device open routine returns an unused minor device number for the stream. Each stream to a clonable device is associated with an unused minor device; therefore, the total number of streams that can be connected to a clonable driver depends on how many minor devices are configured for that driver.

Cloning is often used for protocol pseudo-device drivers that require several streams over which to communicate with users. Upper multiplexors, described in the previous section, are clonable devices. An application has no control over whether a driver supports cloning, because this option is built into the driver. TCP, IP, and UDP are clonable drivers.

Basic STREAMS operations

Applications access STREAMS facilities via standard and STREAMS-specific system calls. Standard system calls used with STREAMS are:

- `open`—Open a stream
- `read`—Read data from a stream
- `write`—Write data to a stream
- `ioctl`—Control a stream
- `close`—Close a stream

STREAMS-specific system calls are:

- `poll`—Notify the user process when selected events occur on a stream
- `getmsg`—Receive a message at the stream head
- `putmsg`—Send a message downstream

This section describes how a stream is opened and is used to transfer data with the `read` and `write` system calls. Using `poll`, `ioctl`, `getmsg`, and `putmsg` is explained in "Programming STREAMS applications" on page 91.

Figure 41 shows an example of opening two devices, writing data to one device, and reading the input back to the other device.

Figure 41 Opening a STREAMS device and transferring data

```
#include <fcntl.h>
#include <stdio.h>

main()
{
    char buf[256];
    int i, fd0, fd1, count;

    /* Open the streams test devices */
    if ((fd0 = open("/dev/strtest0", O_RDWR)) < 0) {
          perror("open of /dev/strtest0 failed");
          exit(1);
    }

    if ((fd1 = open("/dev/strtest1", O_RDWR)) < 0) {
          perror("open of /dev/strtest1 failed");
          exit(1);
    }

    /* Initialize the buffer */
    for (i = 0; i < 256; i++)
          buf[i] = 'a';

    /* Attempt to write the buffer to the device */
    if ((count = write(fd0, buf, 256)) != 256) {
          if (count == -1)
                perror ("write failed");
          else
                fprintf(stderr, "Only %d bytes written\n", count);
          exit(1);
    }

    /* Attempt to read the buffer back */
    if ((count = read(fd1, buf, 256)) != count) {
          if (count == -1)
                perror ("read failed");
          else
                fprintf(stderr, "Only %d bytes read\n", count);
          exit(1);
    }
    exit(0);
}
```

The example in Figure 41 uses two test devices, strtest0 and strtest1. Data written to /dev/strtest0 is passed to /dev/strtest1 for reading. (Data could also be written to /dev/strtest1 and passed to /dev/strtest0 for reading.)

The first step in the example is to open both devices for reading and writing. When opening the devices, the system recognizes strtest0 and strtest1 as STREAMS devices and connects a stream to each device.

A buffer that holds 256 bytes of data is initialized, and the `write` call transmits the contents of the buffer to strtest0. The `read` call sends the number of bytes written to strtest0 to strtest1. The program checks to see if the same number of bytes written to the first device is read back to the second device.

The `exit` system call terminates the process and dismantles the streams by closing all open files.

Service interfaces

Each STREAMS module provides neighboring modules with a set of services and an interface to those services. A well-defined service interface enables modules that support the same interface to interact. In addition, protocols can be substituted below the service interface without having to modify applications.

A service user makes a request of a service provider in the form of a primitive. The service provider also sends primitives to the user for responses and event indications. A service interface is defined as the primitives that define services and the allowable state transitions, or sequence of primitives passed between user and provider. Refer to the section "Interface protocols" on page 109 for more information on service interfaces.

Programming STREAMS applications

5

This chapter describes how to program STREAMS applications to:

- Poll streams for events
- Control modules and drivers
- Handle messages

Input/output polling

The `poll` system call enables a user process to monitor input and output on a set of file descriptors that reference open streams. `poll` is analogous to the `select` system call used on socket descriptors. It identifies the streams available for reading or writing data and checks for events that you specify. Available event types are:

- `POLLIN`—A nonpriority message can be read from the stream head read queue.
- `POLLPRI`—A priority message can be read from the stream head read queue.
- `POLLOUT`—A message can be written to the stream head write queue.

The syntax of the `poll` system call is

```
poll(fds, nfds, timeout)
```

where

`fds` is an array of file descriptors and events to poll.

`nfds` is the number of file descriptors to poll.

`timeout` is the number of milliseconds `poll` waits if no events are pending. `poll` returns when any of the requested events occurs or when the timer expires. If `timeout` is set to 0, `poll` returns immediately; if set to -1, it blocks until one or more of the requested conditions becomes true.

`fds` points to an array of `struct pollfd` structures. `struct pollfd` is defined in <poll.h> and has the following format:

```
struct pollfd {
    int fd;
    short events;
    short revents;
}
```

where:

`fd` specifies the file descriptor to poll.

`events` is a bitmask containing the bitwise inclusive OR of events to poll on that file descriptor.

`revents` is a bitmask set by the system to indicate which of the requested events has occurred.

Figure 42 shows a multi processing example that uses the `poll` system call to check streams for incoming data.

Figure 42 Polling streams for incoming data

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/poll.h>
#include <sys/signal.h>

main()
{
    char buf[256];
    int i, fd0, fd1, count, id;
    /*
     * Open the streams test devices
     */
    if ((fd0 = open("/dev/strtest0", O_RDWR)) < 0) {
            perror("open of /dev/strtest0 failed");
            exit(1);
    }

    if ((fd1 = open("/dev/strtest1", O_RDWR)) < 0) {
            perror("open of /dev/strtest1 failed");
            exit(1);
    }

    /*
     * Set up a reader and a writer process
     */
    switch(id = fork()) {

    case -1:
            perror ("fork failed");
            exit(1);

    case 0:
            writer(fd0, fd1);
    break;

    default:
            reader(fd0, fd1, id);
    break;
    }
    exit(0);
}
```

Figure 42 Polling streams for incoming data (**continued**)

```
writer(fd0, fd1)
        int fd0, fd1;
{
        int i;
        char buf[256];

        /*
         * Initialize the buffer
         */
        for (i = 0; i < 256; i++)
                buf[i] = 'a';
        /*
         * Loop writing the buffer to the devices
         */
        for (i = 0; i < 10; i++) {
                if ((i < 3) || (i >5))
                        if (write(fd0, buf, 256) != 256) {
                                perror ("write failed");
                                exit(1);
                           }
                if (i >= 3)
                        if (write(fd1, buf, 256) != 256) {
                                perror ("write failed");
                                exit(1);
                        }
                /* Wait for a SIGALRM before next write */
                sleep(20);
        }
}

reader(fd0, fd1, id)
        int fd0, fd1, id;
{
        struct pollfd pollfds[2];
        char buf[256];
        int i, j, count;

        /* Initialize pollfds structure */
        pollfds[0].fd = fd0;
        pollfds[1].fd = fd1;
        pollfds[0].events = POLLIN;
        pollfds[1].events = POLLIN;
```

Figure 42 Polling streams for incoming data **(continued)**

```
        /* Attempt to read the buffer back */
        for (i =0; i < 10; i++) {

            /* Allow time for writes to complete */
            sleep(1);
            /*
             * Poll for input, using -1 (infinite) timeout
             */
            if (poll(pollfds, 2, -1) < 0) {
                    perror("poll failed");
                    exit(1);
            }
            for (j = 0; j < 2; j++) {
                    switch(pollfds[j].revents) {

                    case POLLIN: /* Data available for reading */
                          if ((count =
                             read(pollfds[j].fd, buf, 256)) < 0) {
                                perror("read failed");
                                exit(1);
                          } else {
                                printf(%d: "Read %d bytes from
                                          #%d\n", i, count, j);
                          }
                    break;

                    case 0: /* No data on this stream */
                    break;

                    default: /* Error condition */
                          perror("Error event on poll");
                          exit(1);
                    break;
                    }
            }
            /*
             * Signal writer that reads are done and next write
             * can start
             */
            kill(id, SIGALRM);
        }
}
```

In Figure 42, the main routine opens two devices, thereby creating two separate streams. strtest0 and strtest1 are test devices. Everything that is written to either device is available for reading from the other. After opening the devices, the main routine forks to create another process.

Data is written alternately to strtest0 and strtest1 by the child process (writer). It loops 10 times, writing to strtest0 for the first three iterations, to strtest1 for the next 3 iterations, and to both devices for the last four iterations.

The parent process (reader) also loops for 10 iterations and polls each file descriptor for input on each iteration. The `events` field of the `pollfd` structure is set to `POLLIN` to poll for incoming data on each stream.

If `poll` succeeds, the program checks the value of the `revents` field for entry in `pollfds`. If `revents` is set to 0, no event has occurred on that file descriptor; if `revents` is set to `POLLIN`, data is available. Available data is read from the polled device and the results are printed.

To synchronize the two processes, the reader sleeps for 1 second before issuing the `poll` system call to allow the writer to complete the writes. The writer sleeps after each write, waiting for the reader to send a SIGALRM indicating that the data has been received and that the writer can write the next set of data.

Because this example polls only for input, a value other than 0 or `POLLIN` means that an error has occurred. The following error events are defined for `poll` and are only valid for return events:

`POLLERR`	An error message has arrived on the read queue.
`POLLHUP`	A hangup has occurred on the stream associated with the specified file descriptor.
`POLLINVAL`	The requested file descriptor is invalid.

Controlling modules and drivers

STREAMS provides applications with the ability to perform control functions on specific drivers and modules in a stream with `ioctl` system calls. There are three types of STREAMS `ioctl` commands:

- ioctls used to build the streams stack. These include:
  - `I_PUSH`—Insert a module directly below the streamhead
  - `I_POP`—Remove the module directly below the streamhead
  - `I_LINK`—Link a driver beneath a multiplexor
  - `I_UNLINK`—Unlink a driver from a multiplexor

  Users do not usually have to place these ioctls because the `knetdctl` utility takes care of issuing iotcls when it builds the streams stack.
- ioctls interpreted by the streamhead. These include all ioctls except `I_STR`.
- `I_STR`—ioctl used for module and driver-specific ioctl commands. This ioctl causes the streamhead to format a specific ioctl command to be formatted into `M_IOCTL` messages by the streamhead and sent downstream to the appropriate module or driver.

Network devices or connections accessed via a STREAMS `open` call must use the `I_STR` format ioctl for control purposes. Devices or connections accessed via a "socket" call must not use the `I_STR` ioctl format. Programs must not mix ioctl formats.

This section describes using `I_STR` to send ioctl commands to modules and drivers. For more information on the other types of ioctls, refer to the streams(4) man page.

STREAMS drivers use the following format for the `ioctl` system call:

```
#include <sys/ioctl.h>
ioctl(d, request, argp)
int d, request;
char *argp;
```

`ioctl` requests to STREAMS modules and drivers are made indirectly using the `I_STR ioctl` call. The argument to `I_STR` must be a pointer to a `strioctl` structure, which specifies the request. `struct strioctl` is defined in stropts.h and has the format

```
struct strioctl {
    int ic_cmd;         /* command */
    int ic_timout;      /* timeout value */
    int ic_len;         /* length of data */
    char *ic_dp;        /* pointer to data */
}
```

where

`ic_cmd` Identifies the command issued to a module or driver.

`ic_timeout` Specifies the number of seconds an `I_STR` request should wait for an acknowledgment before timing out. A 0 value indicates that the default system timeout value should be used. A positive value indicates that the ioctl must wait that number of seconds before timing out. If a value of -1 is specified, the ioctl does not time out.

`ic_len` Specifies the number of bytes of data to accompany the request.

`ic_dp` Points to the data.

The stream head takes an `I_STR` request and packages it into message from information in the `strioctl` structure. The message is sent downstream to be processed by the first module or driver that understands the request specified by `ic_cmd`. If a module does not recognize the command, it passes the message downstream; if the driver receives an `I_STR` command it does not recognize, it returns a negative acknowledgement.

The `ioctl` call blocks for the amount of time specified by `ic_timeout` until the target module or driver responds with a positive or negative acknowledgement message. If no acknowledgement is received within the specified time, the `ioctl` call fails.

Only one `I_STR` request can be active on a stream at one time. Additional `I_STR` requests block until the active request is acknowledged and the system call completes.

The `strioctl` structure is also used to retrieve any data resulting from an `I_STR` request. If the target module or driver returns data, `ic_dp` must point to a buffer large enough to hold the data, and `ic_len` is set on return to indicate the amount of data returned.

Figure 43 shows an example of using the `I_STR` ioctl. The ioctl in this example is `I_STR_CASE_CHANGE`. This ioctl causes the STREAMS driver to convert all data written to it from lowercase to uppercase.

Figure 43 Sending an ioctl command to a STREAMS module

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/stropts.h>

main()
{
        char                buf[256];
        int                 i,
                            fd0,
                            fd1,
                            count,
                            on = 1;
        struct strioctl stri;


        /* Open the streams test devices */
        if ((fd0 = open("/dev/strtest0", O_RDWR)) < 0) {
                perror("open of /dev/strtest0 failed");
                exit(1);
        }

        if ((fd1 = open("/dev/strtest1", O_RDWR)) < 0) {
                perror("open of /dev/strtest1 failed");
                exit(1);
        }

        /* Format the ioctl */
        stri.ic_cmd = STR_CASE_CHANGE;
        stri.ic_timout = 0;
        stri.ic_len = sizeof(on);
        stri.ic_dp = (char *)&on;

        /* Place the ioctl */
        if (ioctl(fd0, I_STR, (char *)&stri) < 0) {
                perror("ioctl");
                exit(1);
}

        /* Initialize the buffer */
        for (i = 0; i < 256; i++)
                buf[i] = 'a';
```

Figure 43 Sending an ioctl command to a STREAMS module **(continued)**

```
        /* Attempt to write the buffer to the device */
        if ((count = write(fd0, buf, 256)) != 256) {
                if (count == -1)
                        perror ("write failed");
                else
                        fprintf(stderr, "Only %d bytes written\n",
                                        count);
                exit(1);
        }

        /* Attempt to read the buffer back */
        if ((count = read(fd1, buf, 256)) != count) {
                if (count == -1)
                        perror ("read failed");
                else
                        fprintf(stderr, "Only %d bytes read\n", count);
                exit(1);
        }

        printf("The first ten bytes of data are:\n");
        for (i = 0; i < 10; i++)
                putchar(buf[i]);
        putchar('\n');
        exit(0);
}
```


The first step in Figure 43 is to open the test devices strtest0 and strtest1. The next step is to format the `struct strioctl` (the format of `struct strioctl` is shown on page 97):

- The command field (`ic_cmd`) is set to `STR_CASE_CHANGE`.
- The timeout (`ic_timout`) is set to 0 (system default).
- The data pointer (`ic_dp`) is set to point to the data of the ioctl. For `STR_CASE_CHANGE`, the data is an integer (1 is used in the example). When 1 is passed, the driver converts all data written to it from lowercase to uppercase. If a 0 is passed, no case conversion is performed.
- The length of the data (`ic_len`) is set to indicate the length of the data field.

The ioctl is placed. The command is `I_STR`, which takes as its argument a pointer to the `strioctl` structure containing the `STR_CASE_CHANGE` command.

Data is written to strtest0 and read from strtest1. The first 10 bytes of read data, a string of capitalized "A"s, are printed after a successful read.

Message handling

Two STREAMS-specific system calls, `putmsg` and `getmsg`, are used to send messages downstream and receive messages at the stream head. This section explains how to use `getmsg` and `putmsg` to send and receive service primitives and user data across a service interface.

Each service primitive is contained in a STREAMS message that has two parts:

- Control part—Information that identifies the primitive and associated parameters.
- Data part—User data associated with the primitive.

For example, a transport protocol connect request is a service primitive that a transport user sends a transport provider to request a connection with another transport user. Parameters associated with the primitive may be a destination address and options to be used with the connection. The primitive and parameters would be contained in the control part of the message. (The data part would not contain data because TCP does not allow data to be sent with the connect request.)

Two STREAMS-specific system calls, `putmsg` and `getmsg`, are used to pass primitives and data between modules. `putmsg` and `getmsg` are comparable to `write` and `read`, respectively; however, `putmsg` and `getmsg` can pass the control and data parts of a message separately, whereas `write` and `read` are byte-stream oriented and write only data without preserving message boundaries. A service interface requires message boundaries to be preserved so that the beginning and end of each primitive can be located. In addition, `putmsg` and `getmsg` provide separate buffers for the control and data part of a message while `write` and `read` use only one buffer.

Putting a message on the stream

The `putmsg` system call puts a message on the write queue of the next downstream module or driver in a stream. The application specifies the contents of the control and data parts of the message in two separate buffers.

The syntax of `putmsg` is:

```
int putmsg (fd, ctlptr, dataptr, flags)
```

```
int fd;
struct strbuf *ctlptr, *dataptr;
int flags;
```

where

`fd`	Identifies the stream to which the message is passed.
`ctlptr`	Identifies the control part of the message.
`dataptr`	Identifies the data part of the message.
`flags`	May be set to either `0` or `RS_HIPRI`. If set to `0`, `ctlptr` is formatted into a `M_PROTO` message. If set to `RS_HIPRI`, `ctlptr` is formatted into a `M_PCPROTO` message.

Both `ctlptr` and `dataptr` point to a structure of type `strbuf` that defines the size and location of the buffer. `struct strbuf` is defined in stropts.h and has the format

```
struct strbuf {
   int maxlen;     /* not used by putmsg */
   int len;        /* Size of data */
   char *buf;      /* pointer to buffer */
}
```

Messages are created differently based on the value of `len`:

- If `dataptr->len` (or `ctlptr->len`) is -1, no data (or control) portion is created for this message. The same effect is achieved by setting `dataptr` or `ctlptr` to null.
- If `len` is greater than or equal to zero, a `len` byte data or control part is created. If `len` is greater than the message size for the next queue, the message is not sent and an `ERANGE` error is returned.

Retrieving a message from the stream

The `getmsg` system call retrieves the next message from the read queue of a stream and places the contents in two user buffers.

The syntax of `getmsg` is:

```
int getmsg (fd, ctlptr, dataptr, flags)
int fd;
struct strbuf *ctlptr, *dataptr;
int *flags;
```

where

`fd` Identifies the stream from which the message is retrieved.

`ctlptr` Identifies the control part of the message.

`dataptr` Identifies the data part of the message.

`flags` May be set to point to either `0` or `RS_HIPRI`. If set to `0`, `getmsg` attempts to read the first message in the read queue. If set to `RS_HIPRI`, `getmsg` attempts to read a high-priority message. If `getmsg` retrieves a high-priority message, `flags` is set to `RS_HIPRI` on return.

Both `ctlptr` and `dataptr` point to a structure of type `strbuf` that defines the size of the buffer, its location, and how much data was read in. `struct strbuf` has the format

```
struct strbuf {
   int maxlen;     /* size of buffer */
   int len;        /* set on return; amount */
                   /* of data read in */
   char *buf;      /* pointer to buffer */
}
```

Messages are processed differently based on the value of `len`:

- If `dataptr->maxlen` (or `ctlptr->maxlen`) is -1, any data (or control) portion is left on the read queue. `len` is set to -1. The same effect is achieved by setting `dataptr` or `ctlptr` to null.
- If `maxlen` is `0`, a zero length data (or control) message is read. If the message length is greater than zero, the message is left on the queue. In either case, `len` is set to `0` on return. If the message contains no data (or control) portion, `len` is set to `-1`.
- If `maxlen` is greater than zero, up to `maxlen` bytes of data or control information is stored in the buffer. `len` is set to the actual number of bytes retrieved. Any remaining data is requeued. If there is no data (or control) portion, `len` is set to `-1`.

Figure 44 presents an example using `putmsg` and `getmsg` to send and receive data over streams.

Figure 44 Transferring data with putmsg and getmsg

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stropts.h>

main()
{
        struct strbuf  databuf,
                       ctlbuf;
        char           dbuf[20],
                       cbuf[20];
        int            i,
                       fd0, fd1,
                       count;

        /* Open the streams test devices */
        if ((fd0 = open("/dev/strtest0", O_RDWR)) < 0) {
                perror("open of /dev/strtest0 failed");
                exit(1);
        }

        if ((fd1 = open("/dev/strtest1", O_RDWR)) < 0) {
                perror("open of /dev/strtest1 failed");
                exit(1);
        }

        /* Initialize the buffers and the structures */
        for (i = 0; i < 20; i++) {
                dbuf[i] = 'd';
                cbuf[i] = 'c';
        }
        databuf.len = 20;
        databuf.buf = dbuf;
        ctlbuf.len = 20;
        ctlbuf.buf = cbuf;

        /* Attempt to send the buffers */
        if (putmsg(fd0, &databuf, &ctlbuf, 0)< 0) {
                perror ("putmsg failed");
                exit(1);
        }
```

Figure 44 Transferring data with putmsg and getmsg **(continued)**

```
        /* Initialize the buffers for receive */
        bzero(dbuf, sizeof(dbuf));
        bzero(cbuf, sizeof(cbuf));
        databuf.len = 0;
        databuf.maxlen = sizeof(dbuf);
        databuf.buf = dbuf;
        ctlbuf.len = 0;
        ctlbuf.maxlen = sizeof(cbuf);
        ctlbuf.buf = cbuf;

        /* Attempt to receive the buffers */
        if (getmsg(fd1, &databuf, &ctlbuf, 0) < 0) {
                perror ("getmsg failed");
                exit(1);
        }

        if ((databuf.len < 20) || (ctlbuf.len < 20)) {
                fprintf(stderr, "Too few bytes received\n");
                exit(1);
        }

        printf ("First byte of data is %c\n", dbuf[0]);
        printf ("First byte of control is %c\n", cbuf[0]);
        exit(0);
}
```

The example shown in Figure 44 is similar to the example in Figure 41 in Chapter 4, except that `putmsg` and `getmsg` are used to transfer the data instead of `read` and `write`. If `putmsg` places a message that has a data portion and no control portion on the stream, a `read` system call can be used to retrieve the data. The `read` call can not be used to retrieve a message that has a control portion.

After opening the test devices, the buffers and structures are initialized. The data message contains all "d"s, and the control message contains all "c"s.

For `putmsg`, the `len` and `buf` fields are initialized in the `strbuf` structures. For `getmsg`, the `maxlen` and `buf` fields are initialized and the `len` field is set by the system and checked on return from the call.

The message retrieved by `getmsg` should have "d"s in the data portion and "c"s in the control portion.

STREAMS programming for TCP/IP

6

This chapter describes the Transport Provider Interface (TPI) and shows how to send and receive data over TCP/IP using STREAMS system calls and TPI. Programming examples include sending a datagram, establishing a connection, and setting TPI options.

Interface protocols

STREAMS provides the structure for messages to flow up and down a protocol stack and allows for modularity within the protocol implementation. STREAMS also enables networking applications to use standard service interfaces when accessing different protocols.

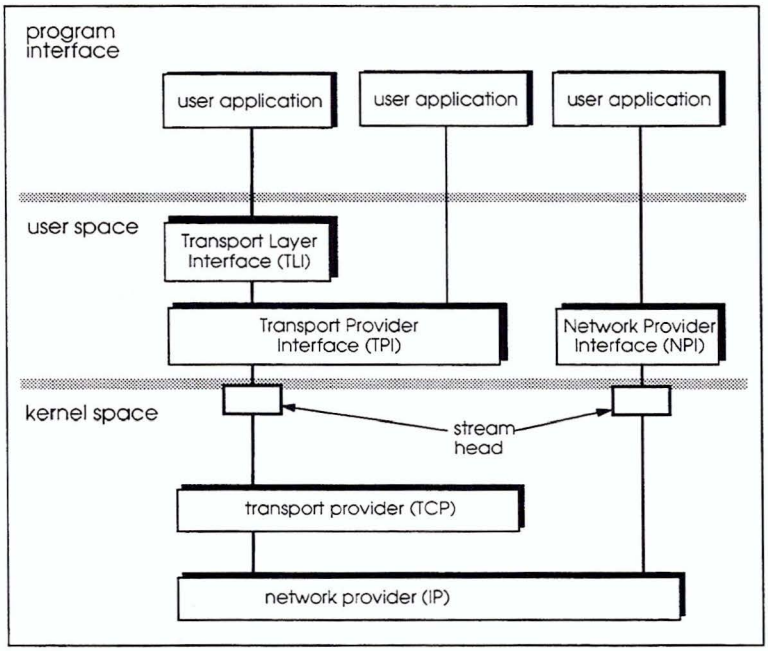
A service interface is a set of primitives and rules for using these primitives that define a service at the boundary between a service user and a service provider. For example, a service interface might define the message format and sequence of messages necessary to establish a networking connection.

The Transport Provider Interface (TPI) is a service interface defined in part by the definition of a standard set of messages. Transport primitives, as defined in the System V Interface Definition (SVID), are messages exchanged between the transport user (networking application) and the transport provider (for example, TCP) using the `getmsg` and `putmsg` system calls. The Transport Layer Interface Library, described in Part III, is a high-level interface to transport layer primitives that hides the actual STREAMS interface from the user. An application using TPI or the TLI library can communicate with the transport layer of any protocol that provides a TPI interface.

The network layer protocol is used to communicate between a transport service provider, such as the TCP driver, and a network service provider, such as the IP driver. Network layer services are accessed by using network primitives defined in the Network Provider Interface (NPI). A user can access the network driver by opening a file descriptor to the driver and passing NPI messages to it.

Figure 45 shows the application program interfaces provided by STREAMS.

Figure 45 STREAMS program interfaces



Prior to STREAMS, applications used sockets to access network services. A socket-to-STREAMS interface is provided with the ConvexOS STREAMS implementation to enable existing socket-based applications to transparently access the STREAMS architecture. The socket-to-STREAMS interface maps socket requests to the transport providers (TCP and UDP) to STREAMS structures and functions. The socket-to-STREAMS interface does not map socket requests to the network provider (IP). New applications can use the TLI library, STREAMS (using TPI or NPI), or sockets. Note that because of the sockets-to-STREAMS conversion, there is some additional overhead in using sockets.

Datagram message example

The example code shown in the next three figures accomplishes the following with STREAMS system calls and transport primitives:

- Establishes a stream to the transport provider and binds a protocol address to the stream
- Sends a datagram to a remote user
- Receives a datagram from a remote user
- Closes the stream connected to the transport provider

Six primitives, defined in tiuser.h, are used in this example. The first two are sent downstream from the user to the provider:

`T_BIND_REQ`	Requests the provider to bind a specified protocol address. The provider must send an acknowledgment.
`T_UNITDATA_REQ`	Requests the provider to send data to the specified address. This primitive does not require an acknowledgment from the provider.

The next four primitives are sent upstream from the provider to the user:

`T_BIND_ACK`	Acknowledges that a previous bind request was successfully processed.
`T_UNITDATA_IND`	Indicates the arrival of a datagram for the user.
`T_ERROR_ACK`	Indicates an error occurred with an earlier request.
`T_UDERROR_IND`	Indicates that received unitdata has an incorrect number of received bytes or bad checksum on the data.

Opening the stream and sending the bind request

The code in Figure 46 opens the stream to the transport provider (udp) and binds an address to it. (This example is not complete; some details are omitted for simplicity. For example, not all data structures are declared and initialized, and there is insufficient error checking.)

Figure 46 Opening the stream and sending the bind request

```
#include <stropts.h>
#include <in.h>
#include <tli.h>

int fd;                                 /* stream descriptor */
struct strbuf ctlbuf;                   /* from stropts.h */
struct strbuf databuf;                  /* from stropts.h */
union T_primitives rcvbuf;              /* from tli.h */

      /* bind request structure for control portion of putmsg */

struct bind_req_t {
   struct T_bind_req bind;              /* space for primitive */
   struct sockaddr_in addr;             /* space for address */
};

struct bind_req_t bind_req;

      if ((fd = open("/dev/udp",O_RDWR)) <0) {/* open the stream to udp */
            perror("open");
            exit(5);
      }

      /* set up bind primitive */

      bind_req.bind.PRIM_type = T_BIND_REQ;
      bind_req.bind.ADDR_length = sizeof(struct sockaddr_in);
      bind_req.bind.ADDR_offset = sizeof(struct T_bind_req);
      bind_req.bind.CONIND_number = 0;

      /* initialize sockaddr struct to bind a default local addr */

      bind_req.addr.sin_family = AF_INET;
      bind_req.addr.sin_port = 0;
      bind_req.addr.sin_addr = 0;

      /* set up control structure */

      ctlbuf.len = sizeof(bind_req);
      ctlbuf.buf = (char *) &bind_req;

      if (putmsg(fd, &ctlbuf, NULL, 0) < 0) {/* send bind request */
            perror("bind");
            close(fd);
            exit(5);
      }
```

Figure 46 Opening the stream and sending the bind request (continued)

```
/* wait for bind ack */
if (get_hipri(fd, &rcvbuf, &ctlbuf) < 0) {
        perror("bind ack");
        close(fd);
        exit(5);
}

/* if not bind ack then exit */

if(rcvbuf.type != T_BIND_ACK){
    if(rcvbuf.type == T_ERROR_ACK) {
            perror("bind ack");
            close(fd);
            exit(5);
    }
    else {
            perror("bind ack");
            close(fd);
            exit(5);
    }
    close(fd);
    exit(1);
}
```


The structure `bind_req_t` describes the contents of the control part of the bind request primitive. After opening the udp driver, the program formats a bind request to send downstream. The first field in the control part defines the type of primitive to be passed.

`putmsg` is called to send the bind request. The bind request message consists of a control part that contains the structure `bind_req`. The `dataptr` argument to `putmsg` is set to `NULL` because the message has no data part. `ctlbuf` is a structure of type `strbuf` (shown in Chapter 2, the section "Putting a message on the stream" on page 102) and is initialized with the primitive type and address. The `maxlen` field of `ctlbuf` is not set because `putmsg` does not use it. The `flags` argument is set to `0`, which means that the message is not a priority message.

After `putmsg` sends the bind request, the acknowledgment of the bind request must be retrieved. The acknowledgment message consists of a control part that contains either a `T_bind_ack` or a `T_error_ack` structure, and no data part. Acknowledgment primitives are defined as priority messages, which means they are placed at the front of the stream head queue ahead of normal messages.

The routine get_hipri is used to retrieve the priority message. get_hipri is defined as

```
get_hipri(fd,rcvbuf,ctlbuf)
int fd;
union T_primitives *rcvbuf;
struct strbuf *ctlbuf;
{
    int flags;

    ctlbuf->maxlen = sizeof(union T_primitives);
    ctlbuf->len = 0;
    ctlbuf->buf = (char *)rcvbuf;
    flags = RS_HIPRI;
    return(getmsg(fd,ctlbuf,NULL,&flags));
}
```

Before calling getmsg, get_hipri initializes the strbuf structure for the control part. buf points to a buffer that can hold the expected control part, and maxlen is set to the maximum number of bytes this buffer can hold.

The dataptr argument to getmsg is set to NULL because neither acknowledgment primitive has a data part. flags is set to RS_HIPRI to indicate that the message to be retrieved is a priority message. getmsg blocks until a priority message arrives.

On return from getmsg, the example checks the primitive type. A T_bind_ack indicates a successful bind operation.

Sending the datagram

The code in Figure 47 sends a datagram to the UDP driver for transmission to the user at the address specified in dest.

Figure 47 Sending the datagram

```
      /* datagram request structure for control portion of putmsg */
struct unit_data {
   struct T_unitdata_req data;          /* primitive portion */
   struct sockaddr_in dest;             /* address portion */
};
char   packet[1000];                    /* space for data */
struct unit_data udr;

      /* setup control portion */
   udr.data.PRIM_type = T_UNITDATA_REQ;
   udr.data.DEST_length = sizeof(struct sockaddr_in);
   udr.data.DEST_offset = sizeof(struct T_unitdata_req);
   udr.data.OPT_length = 0;
   udr.data.OPT_offset = 0;
      /* copy destination address into control
         structure. Assumes (to) was
         setup previously */

   bcopy(&(to->sin_addr), udr.dest, udr.data.DEST_length);

   ctlbuf.len = sizeof(udr);
   ctlbuf.buf = (char *) &udr;

   databuf.len = cc;
   databuf.buf = (char *) packet;

   if (putmsg(fd,&ctlbuf,&databuf,0) < 0) { /* putmsg shouldn't fail */
      close(fd);
      exit(1);
   }
```

The datagram request primitive has both a control part and a data part. The control part contains a `T_unitdata_req` structure that identifies the primitive as a `T_UNITDATA_REQ` and specifies the destination address of the datagram. The data to be sent is placed in the data part of the request message. No acknowledgment needs to be sent in response to a datagram request message.

Receiving the datagram

The last part of this example, shown in Figure 48, retrieves the datagram.

Figure 48 Receiving the datagram

```
struct sockaddr  senders_addr;              /* space for senders address */
char    senders_options[40];                /* space for senders options */

       /* setup for primitive reception, control fields */
       rcvbuf.type = 0;
       ctlbuf.len = 0;
       ctlbuf.maxlen = sizeof(union T_primitives);
       ctlbuf.buf = (char *) &rcvbuf;

       /* setup for data reception,in data fields */
       databuf.len = 0;
       databuf.buf = (char *) packet;
       databuf.maxlen = sizeof(packet);
       flags = 0;

       /* do receive from stream */
       if (getmsg(fd,&ctlbuf,&databuf,&flags) < 0) {
          if (errno == EINTR)
              continue;
       fprintf(stderr, "%s: ", progname);
       perror("getmsg");
       }
       if (rcvbuf.type != T_UNITDATA_IND){
          if (rcvbuf.type == T_UDERROR_IND) {
              fprintf(stderr, "%s: %s\n", progname,
                 perror(rcvbuf.uderror_ind.ERROR_type));
          }
          fprintf(stderr, "%s: Invalid message type received (%x)\n",
              progname, rcvbuf.type);
       }

       /* extract sender's address */
       bcopy(&rcvbuf + sizeof(rcvbuf.unitdata_ind.SRC_offset,
              &senders_addr, rcvbuf.unitdata_ind.SRC_length);

       /* extract sender's options */
       bcopy(&rcvbuf + sizeof(rcvbuf.unitdata_ind.OPT_offset,
              &senders_options, rcvbuf.unitdata_ind.OPT_length);
       {
       close(fd);
       exit(1):
       }
```

`getmsg` retrieves the datagram indication primitive, which contains both a control and data part. The control part, contained in `rcvbuf`, identifies the primitive type (`T_UNITDATA_IND`) and the source address of the datagram sender. The data part contains the actual data.

In `ctlbuf`, `maxlen` is set to indicate the maximum size of the buffer where the control information is to be stored, and `buf` points to that buffer. `databuf` is initialized similarly for the data.

The `flags` argument to `getmsg` is set to zero to indicate that the next message should be retrieved at the stream head, whether normal or priority. If there is no message at the stream head, `getmsg` blocks until a message arrives.

The program checks the type of primitive to make sure it is a datagram indication. If it is not, the system prints an error. The sender's address and options are copied into `senders_addr` and `senders_options` and the stream is closed.

Connection-oriented example

The example code in Figure 50 on page 120 and Figure 51 on page 125 shows using STREAMS system calls and TPI primitives to connect to a host on a TCP/IP network. The basic steps for the connecting side, or client, to connect to a remote host, or server, are

- Establish a stream to the transport provider and bind a protocol address to the stream.
- Send a connect request to the server.
- Receive the connect confirmation from the transport provider.
- Close the stream connected to the transport provider.

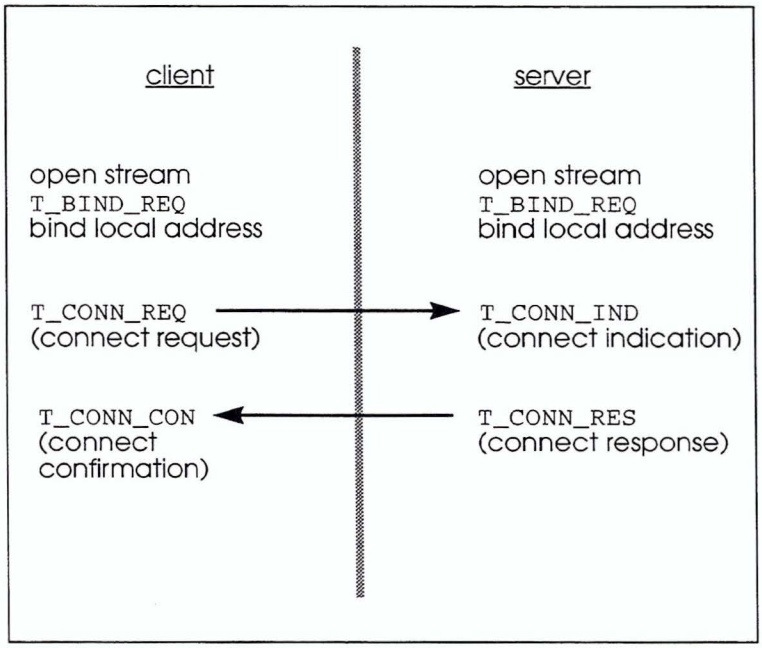
The basic steps for the listening/accepting side to complete the connection are

- Establish a stream on which to listen for incoming connect requests and bind a local address to it.
- Receive a connect indication from the transport provider.
- Send a connect response to the client.

After the client receives the connect confirmation, data can be transmitted.

Figure 49 shows the sequence of steps in establishing a connection between the client and the server.

Figure 49 Connection establishment sequence



Eight primitives, defined in tli.h, are used in Figure 50 and Figure 51. The first three are sent downstream from the user to the transport provider:

`T_BIND_REQ` Requests the transport provider to bind a specified protocol address to the stream and negotiate the number of outstanding connect indications allowed on the stream. The transport provider must acknowledge the request with a `T_BIND_ACK` or `T_ERROR_ACK` primitive (described below).

`T_CONN_REQ` Requests the transport provider to connect to specified destination address. The transport provider must acknowledge the primitive with a `T_OK_ACK` or `T_ERROR_ACK` primitive (described below).

`T_CONN_RES` Requests the transport provider to accept a previous connect request on the specified stream. The transport provider must acknowledge the primitive with a `T_OK_ACK` or `T_ERROR_ACK` primitive (described below).

The next five primitives are sent upstream from the transport provider to the user:

T_BIND_ACK	Indicates to the user that the specified protocol address has been bound to the stream and that the specified number of outstanding connect indications are allowed to be queued by the transport provider for the specified protocol address.
T_CONN_IND	Indicates the arrival of a connect request.
T_CONN_CON	Indicates to the user that a connect request has been confirmed on the specified responding address.
T_OK_ACK	Indicates to the user that the previous user-originated primitive was received successfully by the transport provider.
T_ERROR_ACK	Indicates that a nonfatal error occurred in the last user-originated primitive.

Client side

Figure 50 shows the code for the client or connecting side to establish a connection. (This example is not complete; some details are omitted for simplicity. For example, not all data structures are declared and initialized, and there is insufficient error checking.)

The first step is to define structures for the control part of the `putmsg` system call (refer to the chapter "Programming STREAMS applications" for more information on `putmsg`). The control portion of the STREAMS messages must have space allocated for primitives and addresses.

Figure 50 Establishing a connection—client side

```
#include <sys/tiuser.h>
#include <sys/tli.h>
#include <sys/errno.h>
#include <sys/stropts.h>
#include <sys/socket.h>
#include <sys/fcntl.h>
#include <netinet/in.h>

int fd;                         /* stream descriptor */
struct strbuf ctlbuf;           /* from stropts.h */
struct strbuf databuf;          /* from stropts.h */
union T_primitives rcvbuf;      /* from tli.h */

    /* structure definitions for control portion of putmsg */
struct bind_req_t {
    struct T_bind_req bind;     /* space for primitive */
    struct sockaddr_in addr;    /* space for address */
};

struct conn_req_t {
    struct T_conn_req conn;
    struct sockaddr_in dest;
};

    /* data definitions for putmsg/getmsg calls */
struct bind_req_t bind_req;
struct conn_req_t conn_req;
struct T_conn_con *conn_con;
struct T_error_ack *err;

extern int errno;
main()
{
    struct sockaddr_in remote_addr;
    int     packet[100];
    int     flags;

    /* Open a tcp stream. This example is connection oriented. */

    if ((fd = open("/dev/tcp",O_RDWR)) <0) { /* open the stream to tcp */
          perror("open");
          exit(5);
}
```

Figure 50 Establishing a connection—client side **(continued)**

```
/* Set up bind request. CONIND_number is zero since we are not
 * listening on this stream.
 */

bind_req.bind.PRIM_type = T_BIND_REQ;
bind_req.bind.ADDR_length = sizeof(struct sockaddr_in);
bind_req.bind.ADDR_offset = sizeof(struct T_bind_req);
bind_req.bind.CONIND_number = 0;

/* initialize sockaddr struct to bind a local addr */

bind_req.addr.sin_family = AF_INET;
bind_req.addr.sin_port = 20009; /* just picked a port */
bind_req.addr.sin_addr.s_addr = 0x82a84d04; /* host's ip address */

/* set up control structure */

ctlbuf.len = sizeof(bind_req);
ctlbuf.buf = (char *) &bind_req;

if (putmsg(fd, &ctlbuf, 0, 0) < 0) { /* send bind request */
      perror("bind");
      close(fd);
      exit(5);
}
if (get_hipri(fd, &rcvbuf, &ctlbuf) < 0) {
   printf("connect: Invalid message type received (%d)\n", rcvbuf.type);
      perror("bind ack");
      close(fd);
      exit(5);
}

/* if not bind ack then exit */

if(rcvbuf.type != T_BIND_ACK){
   if(rcvbuf.type == T_ERROR_ACK) {
      err = (struct T_error_ack *)&rcvbuf;
      printf("bind ack error, errno = %d, tli error = %d\n",
          err->UNIX_error, err->TLI_error);
      close(fd);
      exit(5);
   }
```

Figure 50 Establishing a connection—client side (**continued**)

```
    else {
        printf("connect: Invalid message type received (%d)\n",
                rcvbuf.type);
            perror("bind ack");
            close(fd);
            exit(5);
    }
    close(fd);
    exit(1);
}

/* set up connection request */

conn_req.conn.PRIM_type = T_CONN_REQ;
conn_req.conn.DEST_length = sizeof(struct sockaddr_in);
conn_req.conn.DEST_offset = sizeof(struct T_conn_req);
conn_req.conn.OPT_length = 0;
conn_req.conn.OPT_offset = 0;

/* Set up destination address for connect */

conn_req.dest.sin_family = AF_INET;
conn_req.dest.sin_port = 20000;
conn_req.dest.sin_addr.s_addr = 0x82a84d04;

/* set up control structure */

ctlbuf.len = sizeof(conn_req);
ctlbuf.buf = (char *) &conn_req;

if (putmsg(fd, &ctlbuf, 0, 0) < 0) { /* send connect request */
        perror("connect");
        close(fd);
        exit(5);
}

/* setup for buffers for receiving */

rcvbuf.type = 0;
ctlbuf.len = 0;
ctlbuf.maxlen = sizeof(union T_primitives);
ctlbuf.buf = (char *) &rcvbuf;

databuf.len = 0;
databuf.buf = (char *) packet;
databuf.maxlen = sizeof(packet);
```

Figure 50 Establishing a connection—client side **(continued)**

```
    /* wait for OK_ACK from connect request */

    flags = 0;
    if (getmsg(fd,&ctlbuf,&databuf,&flags) < 0) {
        if (errno == EINTR)
        perror("getmsg");
    }
    if (rcvbuf.type != T_OK_ACK){
        printf("connect: Invalid message type received (%d)\n", rcvbuf.type);
    }

    /* Wait for connection confirmation */

    flags = 0;
    if (getmsg(fd,&ctlbuf,&databuf,&flags) < 0) {
        if (errno == EINTR)
        perror("getmsg");
    }
    if (rcvbuf.type != T_CONN_CON){
        printf("connect: Invalid message type received (%d)\n", rcvbuf.type);
    }

    conn_con = (struct T_conn_con *)&rcvbuf;
    bcopy(&rcvbuf + conn_con->RES_offset, &remote_addr,
          conn_con->RES_length);

    /* Test done, close connection */

    close (fd);
}

get_hipri(fd,rcvbuf,ctlbuf)
int fd;
union T_primitives *rcvbuf;
struct strbuf *ctlbuf;
{
int flags;

ctlbuf->maxlen = sizeof(union T_primitives);
ctlbuf->len = 0;
ctlbuf->buf = (char *)rcvbuf;
flags = RS_HIPRI;
return(getmsg(fd,ctlbuf,0,&flags));
}
```

Opening the stream and sending the bind request

In Figure 50, a stream is opened to the transport provider, which in this case is TCP. The structure `bind_req_t` describes the contents of the control part of the bind request primitive. The fourth field in the control part, `CONIND_number`, is the maximum number of outstanding connect indications on this stream. This field is set to zero because this stream will not be used for listening.

The user calls `putmsg` to send the request to the provider to bind a local address to the stream.

After `putmsg` sends the bind request, the acknowledgment of the bind request must be retrieved. The acknowledgment message consists of a control part that contains either a `T_bind_ack` or a `T_error_ack` structure, and no data part. Acknowledgment primitives are defined as priority messages, which means they are placed at the front of the stream head queue ahead of normal messages.

The routine `get_hipri` is used to retrieve the priority message.

Setting up and sending the connect request

The structure `conn_req_t` describes the contents of the control part of the connect request primitive. The control part contains the primitive and the destination address for the connection. `putmsg` is called to send the connect request.

Receiving the connect request acknowledgment and connect confirmation

After sending the connect request, buffers are initialized to receive messages coming from the transport provider. The first message to be received is the `T_OK_ACK` primitive which acknowledges that the connect request was received by the server. The next message to be received is the connect confirmation which confirms that the connection is accepted or rejected. Although the example does not show data transmission, data can be transmitted after the connect confirmation is received.

The last step is to close the stream.

Server side

Figure 51 shows the code for the server or listening side to establish a connection. (This example is not complete; some details are omitted for simplicity. For example, not all data structures are declared and initialized, and there is insufficient error checking.)

Figure 51 Establishing a connection—server side

```
#include <sys/tiuser.h>
#include <sys/tli.h>
#include <sys/errno.h>
#include <sys/stropts.h>
#include <sys/socket.h>
#include <sys/fcntl.h>
#include <sys/ioctl.h>
#include <netinet/in.h>


int fd, nfd;                    /* stream descriptor */
struct strbuf ctlbuf;           /* from stropts.h */
struct strbuf databuf;          /* from stropts.h */
union T_primitives rcvbuf;      /* from tli.h */

   /* bind request structure for control portion of putmsg */
struct bind_req_t {
   struct T_bind_req bind;      /* space for primitive */
   struct sockaddr_in addr;     /* space for address */
};


struct bind_req_t bind_req;
struct T_conn_res conn_resp;
struct T_conn_ind *conn_ind;
struct strfdinsert fdinsert;
struct T_error_ack *err;

extern int errno;


main()
{
       int flags = 0;
       int packet[100];

   /* open connection based stream for listening */

       if ((fd = open("/dev/tcp",O_RDWR)) <0) { /* open the stream to tcp */
              perror("open");
              exit(5);
       }
```

Figure 51 Establishing a connection—server side **(continued)**

```
/* bind local address to stream. Specify a bind queue length of 1
* to handle the listen.
*/

    bind_req.bind.PRIM_type = T_BIND_REQ;
    bind_req.bind.ADDR_length = sizeof(struct sockaddr_in);
    bind_req.bind.ADDR_offset = sizeof(struct T_bind_req);
    bind_req.bind.CONIND_number = 1;

/* initialize sockaddr struct to bind a local addr */

    bind_req.addr.sin_family = AF_INET;
    bind_req.addr.sin_port = 20000;
    bind_req.addr.sin_addr.s_addr = 0x82a84d04;

/* set up control structure */

    ctlbuf.len = sizeof(bind_req);
    ctlbuf.buf = (char *) &bind_req;

    if (putmsg(fd, &ctlbuf, 0, 0) < 0) { /* send bind request */
            perror("bind");
            close(fd);
            exit(5);
    }

/* Get bind acknowledgment. The ack is High Priority */

    if (get_hipri(fd, &rcvbuf, &ctlbuf) < 0) {
        printf("listen: Invalid message type received (%x)\n", rcvbuf.type);
            close(fd);
            exit(5);
    }
/* if not bind ack then exit */

    if(rcvbuf.type != T_BIND_ACK){
        if(rcvbuf.type == T_ERROR_ACK) {
            err = (struct T_error_ack *)&rcvbuf;
            printf("bind ack error, errno = %d, tli error = %d\n",
               err->UNIX_error, err->TLI_error);
            close(fd);
            exit(5);
        }
```

Figure 51 Establishing a connection—server side **(continued)**

```
        else {
            printf("listen: Invalid message type received (%x)\n",
               rcvbuf.type);
            perror("bind ack");
            close(fd);
            exit(5);
        }
        close(fd);
        exit(1);
    }
/* setup for data reception, in data fields */
    rcvbuf.type = 0;
    ctlbuf.len = 0;
    ctlbuf.maxlen = sizeof(union T_primitives);
    ctlbuf.buf = (char *) &rcvbuf;
    databuf.len = 0;
    databuf.buf = (char *) packet;
    databuf.maxlen = sizeof(packet);
    flags = 0;

/*
 * Wait for connection indication
 */

    if (getmsg(fd,&ctlbuf,&databuf,&flags) < 0) {
        perror("getmsg");
    }
    if (rcvbuf.type != T_CONN_IND){
        printf("listen: Invalid message type received (%x)\n", rcvbuf.type);
        close (fd);
        exit (5);
    }
    conn_ind = (struct T_conn_ind *)&rcvbuf;

/* open new stream to tcp */
    if ((nfd = open("/dev/tcp",O_RDWR)) <0) {
        perror("open");
        close (fd);
        exit(5);
    }

/* Set up connection response. This does the accept of the connection,
 * providing a new file descriptor and stream to handle the
 * connection. The connection response is passed in as the
 * ctlbuf portion of a struct strfdinsert. The QUEUE_ptr field is
 * set as part of the I_FDINSERT ioctl, based upon the offset field
 * of the struct strfdinsert.
 */
```

Figure 51 Establishing a connection—server side (**continued**)

```
        conn_resp.PRIM_type = T_CONN_RES;
        conn_resp.QUEUE_ptr = 0;
        conn_resp.SEQ_number = conn_ind->SEQ_number;
        conn_resp.OPT_length = 0;
        conn_resp.OPT_offset = 0;

    /* Set up ioctl to get queue information of new file descriptor.
     * fdinsert.fildes is set to the new file descriptor and passed
     * downstream.
     */
        fdinsert.offset = sizeof(long); /* offset of QUEUE_ptr */
        fdinsert.fildes = nfd;
        fdinsert.flags = 0;
        fdinsert.databuf.len = 0;
        fdinsert.ctlbuf.len = sizeof(struct T_conn_res);
        fdinsert.ctlbuf.buf = (char *) &conn_resp;
        if (ioctl(fd, I_FDINSERT, &fdinsert) < 0) { /* send ioctl */
            perror("fdinsert ioctl failed");
            close(fd);
            close(nfd);
            exit(5);
        }
    /* wait for OK_ACK from connect response */
        flags = 0;
        if (getmsg(fd,&ctlbuf,&databuf,&flags) < 0) {
            if (errno == EINTR)
            perror("getmsg");
        }
        if (rcvbuf.type != T_OK_ACK){
            printf("connect: Invalid message type received (%d)\n", rcvbuf.type);
        }

        close (fd);
        close (nfd);
}
get_hipri(fd,rcvbuf,ctlbuf)
int fd;
union T_primitives *rcvbuf;
struct strbuf *ctlbuf;
{
    int flags;
    ctlbuf->maxlen = sizeof(union T_primitives);
    ctlbuf->len = 0;
    ctlbuf->buf = (char *)rcvbuf;
    flags = RS_HIPRI;
    return(getmsg(fd,ctlbuf,0,&flags));
}
```

Opening the stream and sending the bind request

After defining the structures used for the `putmsg` and `getmsg` system calls, the server opens a stream to the tcp driver to listen for incoming connect requests. A bind request is sent to the transport provider with the `CONIND_number` field of the `bind_req` structure set to 1, meaning that this stream can have one pending connect indication and will be used for listening.

The server gets the bind acknowledgment and initializes the buffers for receiving data. When a connect indication arrives on the stream, the server opens a new stream to accept the connection so that the first stream can continue to be used for listening.

Sending the connect response

The connect response primitive accepts the connection, and provides a new file descriptor and stream to handle the connection.

The connect response primitive is passed to the transport provider in a `I_FDINSERT` ioctl. This ioctl is used to accept a connection on a stream other than the stream on which the connection request arrived. The stream head converts the `I_FDINSERT` ioctl into a message consisting of a protocol block and a data block, and sends the message downstream. The stream head places a reference to the queue of the endpoint module into the transport primitive.

`argp` of the `ioctl` system call specifies a structure of type `strfdinsert`, specified in stropts.h, that has the following format:

```
struct strfdinsert {
   struct strbuf ctlbuf;
   struct strbuf databuf;
   long flags;
   int fildes;
   int offset;
};
```

where:

`ctlbuf`	Information for control message
`databuf`	Information for data message
`flags`	Priority flags for `ctlbuf`
`fildes`	File descriptor of other stream
`offset`	Number of bytes into buffer for stream

The connect response in Figure 51 is passed in as the `ctlbuf` portion of `struct strfdinsert fdinsert`. The `QUEUE_ptr` field of the connect response primitive `conn_resp` is the data structure of the stream on which the connection is accepted. `QUE_ptr` is set as part of the `I_FDINSERT` ioctl, based on the `offset` field of `struct strfdinsert`.

`fdinsert.filedes` is set to the new file descriptor and passed downstream. After the `OK_ACK` from the connect response is retrieved, data can be transmitted over the new stream.

Setting options using TPI

In TPI, options are read or set using a `T_optmgmt_req` followed by a buffer large enough to contain all option data. The format of the options in the buffer is:

```
struct t_option {
   unsigned long len;          /* Length of entire structure */
   unsigned long level;        /* Protocol level affected */
   unsigned long name;         /* Option name */
   unsigned long value[1];     /* The starting byte of the value */
};
```

Each option, even if only one is being set or read, uses this structure. Fields in the `t_option` structure are:

`len` Describes in words the length of the `t_option` structure for this option. For example, when setting the option `SO_REUSEADDR`, the `len` field would be 4 (`len` is 1 word, `level` is 1 word, `name` is 1 word, and `value` fits in 1 word). The length for the `t_option` structure for the option `SO_KEEPALIVE` would be 5, since the keepalive option structure is 2 words long.

`level` Identifies the layer to which the option is directed, such as `INET_GENERIC` (socket level), `INET_UDP` (UDP protocol option), or `INET_IP` (IP level directive).

`name` Identifies the option requested.

`value` Is at least one word; the actual size depends on what `len` is set to. value is defined as an array of 2 elements only to illustrate that is can be more than one word (it could be 5 words).

Options are defined in <sys/socket.h>. Table 4 lists options currently supported by Internet Services protocols.

Table 4 Options supported by Internet Services protocols

Option level	Option name	Used by TCP	Used by UDP
INET_GENERIC	SO_BROADCAST	x	x
	SO_DEBUG	x	x
	SO_DONTROUTE	x	x
	SO_LINGER	x	
	SO_KEEPALIVE	x	
	SO_RCVBUF	x	x
	SO_SNDBUF	x	x
	SO_SNDLOWAT	x	x
	SO_RCVLOWAT	x	x
	SO_REUSEADDR	x	x
	SO_USELOOPBACK	x	x
INET_TCP	TCP_NODELAY	x	
	TCP_MAXSEG	x	
INET_UDP	UDPCHECKSUM		x
INET_IP	IP_TOS	x	x
	IP_TTL	x	x
	IP_OPTIONS	x	x

The programming example in Figure 52

- Opens a stream to TCP
- Binds default port and address to stream
- Reads the default option settings for the stream
- Sets one option on the stream
- Sets two options on the stream with one request
- Reads the current options settings for the stream
- Closes the stream

This example does not validate the return results other than checking for the proper acknowledgment. There is minimal error checking.

The same type of program can be used for either UDP or TCP.

Figure 52 Setting options

```
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/stropts.h>
#include <sys/tiuser.h>
#include <sys/tli.h>
#include <netinet/tcp.h>

#define MAX_LEN 100

extern errno;

char rcvbuf[500];
union T_primitives *rbuf;

struct {
    struct T_bind_req bind;
    struct sockaddr_in addr;
    } bindreq;                    /* bind request */

struct {
    struct T_optmgmt_req req;
    char options[MAX_LEN];
    } optreq;                     /* optmgmt request */

struct {
    struct T_optmgmt_req req;
    char options[MAX_LEN];
    } optret;                     /* optmgmt return */

main()
{
        struct t_option *topt;
        int fd, flags;
        struct strbuf databuf, ctlbuf;

        rbuf = (union T_primitives *) rcvbuf;
        /* open stream to tcp */

        fd = open("/dev/tcp", O_RDWR);
        if (fd == -1) {
            printf("open error %d\n", errno);
            exit (1);
        }
```

Figure 52 Setting options **(continued)**

```
/* bind stream */

bindreq.bind.PRIM_type = T_BIND_REQ;
bindreq.bind.ADDR_length = sizeof(struct sockaddr_in);
bindreq.bind.ADDR_offset = sizeof(struct T_bind_req);
bindreq.bind.CONIND_number = 1;

bindreq.addr.sin_family = AF_INET;
bindreq.addr.sin_port = 0;          /* let system give us a port */
bindreq.addr.sin_addr.s_addr = 0;   /* default to any address */

ctlbuf.len = sizeof(bindreq);
ctlbuf.buf = (char *) &bindreq;

if(putmsg(fd, &ctlbuf, 0, 0) < 0) {
    printf("bind error %d\n", errno);
    close(fd);
    exit(2);
}

/* get bind acknowledgment */

flags = RS_HIPRI;
ctlbuf.maxlen = sizeof(rcvbuf);
ctlbuf.len = 0;
ctlbuf.buf = (char *)&rcvbuf;
if (getmsg(fd, &ctlbuf, 0, &flags) < 0) {
    printf("bind ack getmsg failed, %d\n", errno);
    close(fd);
    exit(6);
}
if (rbuf->type != T_BIND_ACK) {
    printf("bind failed to ack, primitive = %d\n", rbuf->type);
    close(fd);
    exit(7);
}


/* get default option list
 *
 * By setting OPT_length to 0, all default options for the protocol
 * are returned. It is up to the user to insure there is enough
 * buffer space to receive the defaults. Sizes are:
 *
 * TCP - sizeof(struct T_optmgmt_req) + (13 * sizeof(struct t_option)
 *
 * UDP - sizeof(struct T_optmgmt_req) + (12 * sizeof(struct t_option)
 */
```

Figure 52 Setting options **(continued)**

```
optreq.req.PRIM_type = T_OPTMGMT_REQ;
optreq.req.OPT_length = 0;
optreq.req.OPT_offset = sizeof(struct T_optmgmt_req);
optreq.req.MGMT_flags = T_DEFAULT;

ctlbuf.len = sizeof(optreq);
ctlbuf.buf = (char *) &optreq;

if(putmsg(fd, &ctlbuf, 0, 0) < 0) {
    printf("get default error %d\n", errno);
    close(fd);
    exit(2);
}

/* get optmgmt acknowledgment
 * Option management requests return an acknowledgment, which is
 * High priority. Return type should be T_OPTMGMT_ACK.
 */

flags = RS_HIPRI;
ctlbuf.maxlen = sizeof(rcvbuf);
ctlbuf.len = 0;
ctlbuf.buf = (char *)&rcvbuf;
if (getmsg(fd, &ctlbuf, 0,&flags) < 0) {
    printf("optmgmt ack getmsg failed, %d\n", errno);
    close(fd);
    exit(6);
}
if (rbuf->type != T_OPTMGMT_ACK) {
    printf("optmgmt 1 failed to ack, primitive = %d\n", rbuf->type);
    close(fd);
    exit(8);
}

/* set a single option
 *
 * The t_option structure is assumed to be immediately following the
 * T_optmgmt_req structure in memory.
 */
optreq.req.PRIM_type = T_OPTMGMT_REQ;
optreq.req.OPT_length = sizeof(struct t_option);
optreq.req.OPT_offset = sizeof(struct T_optmgmt_req);
optreq.req.MGMT_flags = T_NEGOTIATE;
```

Figure 52 Setting TPI options **(continued)**

```
topt = (struct t_option *)optreq.options;
/*len is set to (3 * sizeof(unsigned long)) + sizeof (value) */
topt->len = 4;
topt->level = INET_GENERIC;
topt->name = SO_REUSEADDR;
topt->value[0] = T_YES;

ctlbuf.len = sizeof(struct T_optmgmt_req) + sizeof(struct t_option);
ctlbuf.buf = (char *) &optreq;

if(putmsg(fd, &ctlbuf, 0, 0) < 0) {
   printf("single option error %d\n", errno);
   close(fd);
   exit(3);
}

/* get optmgmt acknowledgment */

flags = RS_HIPRI;
ctlbuf.maxlen = sizeof(rcvbuf);
ctlbuf.len = 0;
ctlbuf.buf = (char *)&rcvbuf;
if (getmsg(fd, &ctlbuf, 0,&flags) < 0) {
   printf("optmgmt ack getmsg failed, %d\n", errno);
   close(fd);
   exit(6);
}
if (rbuf->type != T_OPTMGMT_ACK) {
   printf("optmgmt 2 failed to ack, primitive = %d\n", rbuf->type);
}

/* set multiple options */

optreq.req.PRIM_type = T_OPTMGMT_REQ;
optreq.req.OPT_length = 2 * sizeof(struct t_option);
optreq.req.OPT_offset = sizeof(struct T_optmgmt_req);
optreq.req.MGMT_flags = T_NEGOTIATE;

topt = (struct t_option *)optreq.options;
/*len is set to (3 * sizeof(unsigned long)) + sizeof (value) */
topt->len = 4;
topt->level = INET_GENERIC;
topt->name = SO_DEBUG;
topt->value[0] = T_YES;
```

Figure 52 Setting TPI options **(continued)**

```
topt = (struct t_option *)((caddr_t)(topt) + sizeof(struct t_option));
/*len is set to (3 * sizeof(unsigned long)) + sizeof (value) */
topt->len = 4;
topt->level = INET_TCP;
topt->name = TCP_NODELAY;
topt->value[0] = T_YES;
/*
 * set the size of the control structure to reflect 2 t_option
 * structures and 1 T_optmgmt_req structure
 */

ctlbuf.len = sizeof(struct T_optmgmt_req) +
    (2 * sizeof(struct t_option));
ctlbuf.buf = (char *) &optreq;

if(putmsg(fd, &ctlbuf, 0, 0) < 0) {
    printf("multiple option error %d\n", errno);
    close(fd);
    exit(4);
}

/* get optmgmt acknowledgment */
flags = RS_HIPRI;
ctlbuf.maxlen = sizeof(rcvbuf);
ctlbuf.len = 0;
ctlbuf.buf = (char *)&rcvbuf;
if (getmsg(fd, &ctlbuf, 0,&flags) < 0) {
    printf("optmgmt ack getmsg failed, %d\n", errno);
    close(fd);
    exit(6);
}
if (rbuf->type != T_OPTMGMT_ACK) {
    printf("optmgmt 3 failed to ack, primitive = %d\n", rbuf->type);
}

/* get current option list
 *
 * By setting OPT_length to 0, all current options for the protocol
 * are returned. It is up to the user to insure there is enough
 * buffer space to receive the options. Sizes are:
 *
 * TCP - sizeof(struct T_optmgmt_req) + (13 * sizeof(struct t_option)
 *
 * UDP - sizeof(struct T_optmgmt_req) + (12 * sizeof(struct t_option)
 */
```

Figure 52 Setting TPI options (**continued**)

```
        optreq.req.PRIM_type = T_OPTMGMT_REQ;
        optreq.req.OPT_length = 0;
        optreq.req.OPT_offset = sizeof(struct T_optmgmt_req);
        optreq.req.MGMT_flags = T_CURRENT;

        ctlbuf.len = sizeof(optreq);
        ctlbuf.buf = (char *) &optreq;

        if(putmsg(fd, &ctlbuf, 0, 0) < 0) {
            printf("current options error %d\n", errno);
            close(fd);
            exit(5);
        }

        /* get optmgmt acknowledgment */

        flags = RS_HIPRI;
        ctlbuf.maxlen = sizeof(rcvbuf);
        ctlbuf.len = 0;
        ctlbuf.buf = (char *)&rcvbuf;
        if (getmsg(fd, &ctlbuf, 0,&flags) < 0) {
            printf("optmgmt ack getmsg failed, %d\n", errno);
            close(fd);
            exit(6);
        }
        if (rbuf->type != T_OPTMGMT_ACK) {
            printf("optmgmt 4 failed to ack, primitive = %d\n", rbuf->type);
        }

        close(fd);
        exit(0);
}
```

Converting raw socket applications to STREAMS

7

The raw socket interface to the IP protocol in no longer supported. ConvexOS now provides a STREAMS interface to IP conforming to the NPI standard. To convert existing raw socket applications to STREAMS/NPI, the following modifications must be made.

First, all calls to socket specifying `SOCK_RAW` as the type, must be converted to a call to `open()`. If an application has the following call:

```
int sock_raw;
int proto;
    if ((sock_raw = socket(AF_INET, SOCK_RAW, proto)) < 0) {
        perror("socket");
        exit(1);
    }
```

It would be replaced by the code segment shown in Figure 53.

Figure 53 Opening the raw IP device

```
#include <sys/stropts.h>
#include <sys/npi.h>
#include <netinet/in.h>
#include <fcntl.h>

struct strbuf ctlbuf;
union N_primitives rcvbuf;
int strm_raw;
int flags;
int proto;
struct {
    N_bind_req_t bind;
    int          addr;
} bind_req;


    /*
     * Open a file descriptor for the STREAMS IP device driver.
     */
    strm_raw = open("/dev/ip", O_RDWR);
    if (strm_raw < 0) {
       perror("opening raw ip device");
       exit(1);
    }

    /*
     * Using putmsg, send an NPI primitive to bind the stream
     * to the ICMP protocol. Using getmsg, receive the acknowledgement
     * of the successful bind.
     */

    bind_req.bind.PRIM_type = N_BIND_REQ;
    bind_req.bind.ADDR_length = sizeof(int);
    bind_req.bind.ADDR_offset = sizeof(N_bind_req_t);
    bind_req.bind.CONIND_number = 0;
    bind_req.bind.BIND_flags = 0;
    bind_req.addr = proto;

    ctlbuf.len = sizeof(bind_req);
    ctlbuf.buf = (char *)&bind_req;

    if (putmsg(strm_raw, &ctlbuf, (struct strbuf *)0, 0) < 0) {
       perror("raw ip bind");
       exit(1);
    }
```

Figure 53 Opening the raw IP device **(continued)**

```
    flags = RS_HIPRI;
    ctlbuf.maxlen = sizeof(rcvbuf);
    ctlbuf.buf = (char *)&rcvbuf;
    if (getmsg(strm_raw, &ctlbuf, (struct strbuf *)0, &flags) < 0) {
        perror("receiving bind acknowledgement");
        exit(1);
    }

    /*
     * Check that the recieved message was a bind acknowledgement.
     */
    if (rcvbuf.type != N_BIND_ACK) {
        perror("Didn't receive bind acknowledgement");
        exit(1);
    }
```


Furthermore, all calls to `send()` and `sendto()` must be converted. If the original socket application had the following code:

```
int rtn,
    sock_raw;
charbuffer[BUFSIZE];
struct sockaddr_in dest;

    rtn = sendto(sock_raw, (char *)buffer, BUFSIZE, 0, &dest, sizeof(dest));
    if (rtn < 0) {
        perror("sendto");
        exit(1);
    }
```

It would be replaced with the code segment shown in Figure 54.

Figure 54 Sending data to the raw IP device

```
#include <sys/npi.h>
#include <sys/stropts.h>

int rtn,
    strm_raw,
    flags = 0;
charbuffer[BUFSIZE];
struct sockaddr_in dest;
struct strbufctlbuf,
       databuf;
struct {
    N_unitdata_req_tdata;
    int       dest;
} udr;

    /*
     * Create NPI unitdata request to set the packet's destination address.
     */
    udr.data.PRIM_type = N_UNITDATA_REQ;
    udr.data.DEST_length = sizeof(dest.sin_addr);
    udr.data.DEST_offset = sizeof(N_unitdata_req_t);
    udr.data.OPT_length = 0;
    udr.data.OPT_offset = 0;
    bcopy(&(dest.sin_addr), &udr.dest, sizeof(dest.sin_addr));

    /*
     * Set the control buffer to point to the unitdata header. Set the
     * data buffer to indicate the data. Send the message using putmsg.
     */
    ctlbuf.len = sizeof(udr);
    ctlbuf.buf = (char *)&udr;
    databuf.len = BUFSIZE;
    databuf.buf = buffer;

    rtn = putmsg(strm_raw, &ctlbuf, &databuf, 0);
    if (rtn < 0) {
       perror("putmsg");
       exit(1);
    }
```

Finally, all calls to `recv()` and `recvfrom()` must be converted. If the original socket application had the following code:

```
int rtn,
    sock_raw;
charbuffer[BUFSIZE];
struct sockaddr_in from;

    rtn = recvfrom(sock_raw, (char *)buffer, BUFSIZE, 0,
        (struct sockaddr *)&from, sizeof(from));
    if (rtn < 0) {
       perror("recvfrom");
       exit(1);
    }
```

It would be replaced with the code segment shown in Figure 55.

Figure 55 Receiving the message

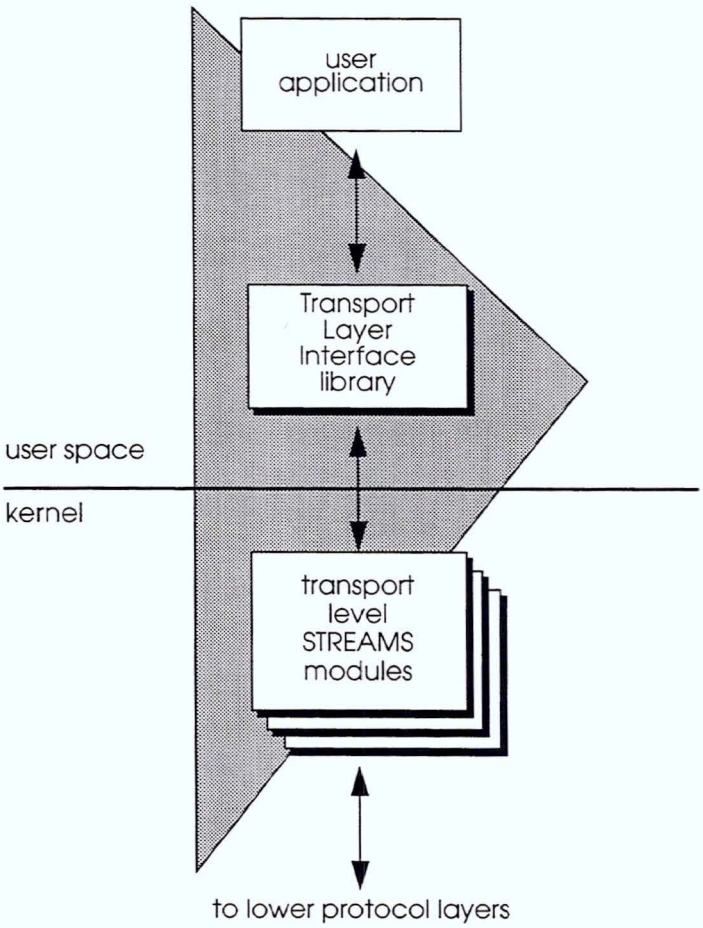
```
#include <sys/npi.h>
#include <sys/stropts.h>

int     rtn,
        strm_raw,
        flags = 0;
char    buffer[BUFSIZE];
struct sockaddr_in from;
struct strbuf ctlbuf,
        databuf;
union N_primitives rcvbuf;

        /*
         * Initialize the control message to indicate the NPI
         * N_primitives union and the data message to indicate
         * the buffer.
         */
        rcvbuf.type = 0;
        ctlbuf.len = 0;
        ctlbuf.maxlen = sizeof(rcvbuf);
        ctlbuf.buf = (char *)&rcvbuf;
        databuf.len = 0;
        databuf.maxlen = BUFSIZE;
        databuf.buf = (char *)buffer;

        /*
         * Receive the message using getmsg, verify that it is
         * a N_UNITDATA_IND, and extract the address if available.
         */
        rtn = getmsg(strm_raw, &ctlbuf, &databuf, &flags);
        if ((rtn < 0) || (rcvbuf.type != N_UNITDATA_IND))
           return(-1);
        if (rcvbuf.unitdata_ind.SRC_length)
           bcopy((char *)&rcvbuf + rcvbuf.unitdata_ind.SRC_offset,
            &from.sin_addr, rcvbuf.unitdata_ind.SRC_length);
```

Transport Layer Interface (TLI) library



Transport Layer Interface (TLI) library

This part describes the interface between a user application and a CONVEX transport-level product. The interface, and the library that implements it, allow you to use transport-level services to exchange data with another user. The following chapters describe:

- Important concepts and operational characteristics
- Transport states, events, and options

This part is written for application programmers who are experienced developers of C-language programs under ConvexOS and are familiar with STREAMS module interfaces and transport interface definitions.

Throughout these chapters, references are made to fields and structures that are discussed in detail in the TLI man pages.

This part is organized into the following chapters:

- Chapter 8 explains general concepts necessary to use the TLI library.
- Chapter 9 introduces the sequences and procedures for using connection-mode and connectionless-mode operation.
- Chapter 10 describes the events that occur on the user-to-transport-level path and the interface states as seen by the user application.
- Chapter 11 contains a programming example.
- Chapter 12 lists the differences between the CONVEX Transport Layer Interface and related standards.
- Chapter 13 describes the option information buffer format for the Internet Services transport layer.

Introduction to the TLI library

8

Overview

The CONVEX Transport Layer Interface (TLI) library is a set of user-level functions that provide high-level access to underlying transport providers.

CONVEX transport providers are implemented as STREAMS modules. The TLI library interacts directly with these products by issuing system calls to the CONVEX STREAMS I/O interface. The library relieves the user of:

- Dealing with the complexities of the transport layer
- Using transport provider interface primitives
- Waiting for acknowledgments

The CONVEX Transport Layer Interface (TLI) defines a set of operations, events, and state transitions for access to transport providers. The interface is, for the most part, independent of specific transport providers. Certain buffer formats are implementation-dependent. The option information buffer for the Internet Services Transport layer is described in "Option information buffer format" on page 199. Refer to the *CONVEX OSI WAN Transport Programmer's Guide* for a description of buffer formats specific to the CONVEX OSI WAN Transport.

Note

Throughout this book, the term *network* refers to the network layer of the OSI model.

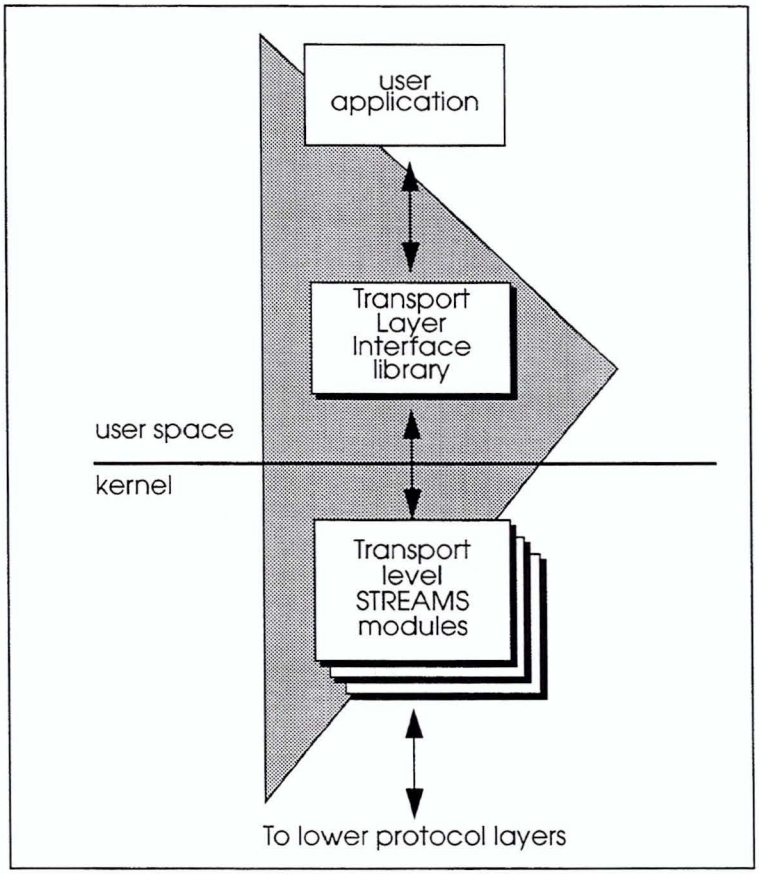
Using the TLI library, a user application can exchange data with a peer application across a CONVEX network or any transport-level network protocol. Various services and options support a wide range of applications from long-term, static exchanges to short-term, dynamic interactions.

The interface is based on two widely-used transport-layer interface standards: System V Interface Definition (SVID) and

X/Open Transport Interface (XTI). The CONVEX TLI library is an extension of these standards. Refer to "Standards compliance of the CONVEX TLI" on page 195 for a description of their differences.

Figure 56 illustrates the relationship between a local user application, the TLI library, and the lower protocol layers.

Figure 56 Relationship of TLI library to user application



Terms and concepts described in this chapter include:

- Transport provider, user, and endpoint
- Service modes
- Error processing
- Execution modes
- Managing events

When appropriate in this and subsequent chapters, descriptions of operations, procedures, sequences, and events include references to specific library functions.

Terms and definitions

A *transport provider* implements the services of the transport-level protocol. In a network defined by the OSI model, the transport layer may include several transport providers, each offering a different type of service. For example, one transport provider may offer wide area network service, while another operates over a local area network. Each transport provider is selected by higher-level applications that use a unique identifier.

A *transport user* uses the services of a transport provider to exchange data across a network defined by the OSI model with a peer application.

The path of communication between a transport user and a specific provider is called a *transport endpoint*. The transport user opens the endpoint using the `t_open()` library routine, which returns a file descriptor, `fd`, identifying the endpoint and other protocol information. Once the endpoint is opened, the user includes the file descriptor in subsequent library function calls to pass requests to the associated transport provider.

After opening an endpoint, the user binds it to a protocol address using the `t_bind()` library routine, making the endpoint available for data exchange. An application may use an endpoint to listen for incoming connect requests rather than for sending or receiving data. To accomplish this, `t_bind()` must be called with the appropriate parameter settings. An endpoint initialized in this manner is said to be bound for listening. Refer to the section "Initializing and deinitializing an endpoint" on page 164 for details.

Two peer users can associate their respective endpoints to form a *connection* for reliable, sequential data exchange. Each transport endpoint supports one connection at a time.

The Transport Layer Interface is a state-driven interface. The transport provider implements the state machine, processes any request supported by its current state, and rejects requests that are not compatible with its current state.

Benefits for developers of transport users include:

- Several processes can share a single transport endpoint. This can be done by forking a child process after opening a transport endpoint. Cooperating processes must synchronize activity on an endpoint to avoid violating its current state.

- A user process can establish an arbitrary number of transport connections subject to the availability of system resources. Separate calls to `t_open()` are made for each connection. These calls do not have to be to the same provider.

Service modes

The Transport Layer Interface offers two service modes:

- Connection-mode
- Connectionless-mode

An individual transport endpoint supports one service mode at a time.

Connection-mode

In *connection-mode* service, the interface logically links the endpoints of two peer transport users. Think of connection mode as a service in which all data exchanged over an established connection is logically related, and end-to-end delivery is guaranteed. This mode is ideal for long, static data exchanges.

Features of connection-mode service include:

- Independent parameters and options for each connection
- Reduced address management overhead during data transfer
- Logical association of data exchanged between peer transport users

The user who initiates a connection is called a *client*. A *server* waits for a client to initiate a connection. Depending on the application, the server may also perform other operations based on requests from the client.

Either user can terminate an established connection in one of two ways: by using the `t_snddis` library routine, or by initiating an orderly release.

- Using `t_snddis()`
  Using the `t_snddis()` function, a user abruptly terminates the connection. An abrupt termination can result in the loss of user data.
- *Initiating orderly release*
  Orderly release is a cooperative way of agreeing that data exchange is complete and terminating a connection without

data loss. Either user can initiate an orderly release, but both users must agree to an orderly release before the transport provider terminates the connection.

Refer to "TLI library functions" on page 159 for more information on connection-mode service.

Connectionless-mode

Using *connectionless-mode*, two users exchange logically unrelated data. This service mode eliminates the overhead of connection establishment, but requires that a destination address be supplied with each unit of data. It offers higher performance for short-lived, highly redundant applications that can be reconfigured dynamically and do not require sequenced, guaranteed delivery of data.

Connectionless-mode service features include:

- Logically independent units of data
- Prior agreement between peer transport users on data transfer parameters and options
- Single service request to transfer a unit of data and all necessary control information
- No relationship between multiple service requests
- Independent routing for each self-contained unit of data

Refer to "TLI library functions" on page 159 for more information on connectionless-mode service.

Error Processing

When a library function detects an error, the function:

- Sets the return value to `-1`
- Stores an error code in external integer `t_errno`, which should be declared in the user's application

Users can call a TLI diagnostic function, `t_error()`, to print an error message based on the returned error code.

Note

After an error occurs, the error code in `t_errno` is not cleared on subsequent library calls. Use the return value, not `t_errno`, to initially check for errors on return from function calls. If an error is indicated, then check `t_errno` for error type.

Two types of errors can occur as part of a library function:

- *Library error* — The error occurred within a library function.
- *System error* — The error is detected on return from a ConvexOS system call. In addition to the normal error notification, the function sets `t_errno` to `TSYSERR`. In general, if a system call invoked by a TLI routine fails, `t_errno` is set to `TSYSERR`. External variable `errno` contains the appropriate system error code.

In most cases, a system error renders the affected endpoint unusable. To recover an unusable endpoint from such an error, a user process must close, reopen, and reinitialize the endpoint.

Execution modes

To allow for the widest range of transport user applications, TLI supports two execution modes: synchronous and asynchronous.

The default execution mode is synchronous. It is particularly useful for event-driven user processes that maintain a single transport connection.

In *synchronous mode*, a library function does not resume execution until the requested operation is complete. The user cannot perform any other operation until execution control is returned. For example, a user might call `t_rcv()` in synchronous mode to retrieve incoming data. The user process resumes execution after the data arrives.

Asynchronous mode allows a user to call a library function and resume execution immediately, whether or not the function is able to complete its processing. For example, assume a user process manages multiple transport connections and calls `t_rcv()` in asynchronous mode to retrieve incoming data from one connection. The function sets `t_errno` to `TNODATA` and returns `-1` if data has not arrived. Separate calls to `t_rcv()` are made for each connection. Refer to "TLI programming example" on page 189 for an asynchronous programming example.

Asynchronous mode execution allows a user process to:

- Perform various operations between events
- Manage multiple connections concurrently
- Efficiently use long delays between activities on any given connection

The setting of the O_NONBLOCK flag determines the mode in effect for a particular transport endpoint. If the flag is set, asynchronous mode is in effect; otherwise, synchronous mode is in effect. The O_NONBLOCK flag is cleared by default. The flag can be set in one of two ways:

- In the t_open() function call
- In the ConvexOS fcntl() system call

Refer to the man page for details. In both instances, the mode applies to subsequent library function calls on the given endpoint until you change the mode with another fcntl() request.

Refer to "Events and states" on page 175 for information on events, asynchronous events, inbound services, and consuming functions that clear events.

TLI library functions

9

Overview

CONVEX TLI offers two types of service modes: connection mode and connectionless mode.

- Connection-mode service enables peer users to exchange data over an established connection. This mode of data transfer is characterized by reliable, sequenced data exchange.
- Connectionless-mode service involves the exchange of relatively short, unrelated messages.

The choice of service mode depends on application requirements. However, library function support and sequence of operations vary depending on the service mode.

The TLI library will not accept a request that violates its state, regardless of service mode. Refer to "Events and states" on page 175 for more information on states.

This chapter describes the library routines for both service modes, including the sequence in which the routines are called. The last section in this chapter describes utility functions.

TLI library routines fall into three groups: routines that are available for either connection-mode service, connectionless-mode service, or both services. Table 5 provides a summary of TLI functions, designates which are utilities, and indicates the type(s) of service mode available.

Table 5 TLI library functions

Routine	Connection mode	Connectionless mode	Utility
t_accept()	x		
t_alloc()			x
t_bind()	x	x	
t_close()	x	x	
t_connect()	x		
t_error()			x
t_free()			x
t_getinfo()			x
t_getstate()			x
t_listen()	x		
t_look()			x
t_open()	x	x	
t_optmgmt()			x
t_rcv()	x		
t_rcvconnect()	x		
t_rcvdis()	x		
t_rcvrel()	x		
t_rcvudata()		x	
t_rcvuderr()		x	
t_snd()	x		
t_snddis()	x		
t_sndrel()	x		
t_sndudata()		x	
t_sync()			x
t_unbind()	x	x	
t_setpeer		x	

Utilities, available with any mode, can be called at any time. Other utility functions provide general information to the user or allow the user to manage memory used for library function calls. These utility functions are independent of the service mode.

Connection-mode functions

Connection-mode functions have five phases:

- Initializing an endpoint
- Establishing a connection
- Transferring data
- Terminating a connection
- Deinitializing an endpoint

To use connection-mode service, initialize a transport endpoint and connect to a remote user. Initialization consists of opening the endpoint and binding it to a protocol address. Refer to the t_open(3) and t_bind(3) man pages for details.

Users exchange data over the connection. When the exchange is complete and the connection is no longer needed, you or another user terminates the connection. Then each user can deinitialize (unbind and close) an endpoint. As an alternative, you can rebind (unbind and bind) an endpoint to another protocol address and initiate another connection.

A user process that manages multiple endpoints typically uses one endpoint to receive and process multiple connection requests. Each connection is established on a different local endpoint. Refer to the t_accept(3) man page for details. The transport provider queues outstanding incoming connection requests, thus allowing you to retrieve them one at a time.

For more information on establishing connection-mode service, refer to the description of a listening application on page 153, and to the section "Establishing a connection" on page 165.

Table 6 illustrates a typical sequence of function calls by a client and a server to establish connection-mode service.

Table 6 Function calls for connection-mode service

Client	Server
1. t_open()	t_open()
2. t_bind()	t_bind()
3.	t_listen()
4. t_connect()	
5.	t_accept()
6. t_snd()	
7.	t_rcv()
8. t_snddis()	
9.	t_rcvdis()
10. t_unbind()	t_unbind()
11. t_close()	t_close()

This table does not include all the functions you might call; rather, the functions are those critical to connection-mode service. An empty space in a column for either user indicates a function call made by the other user prior to the specified user requesting the next function.

As shown in Table 6, the client and server initialize their respective endpoints (lines 1 and 2). The server waits (line 3) for the client to request a connection (line 4). After the server accepts the connection (line 5), either side may begin sending data. The table illustrates data exchange with the client sending and the server receiving (lines 6 and 7). However, the data exchange may consist of multiple send and receive operations initiated by either user. When the data exchange is complete, the client abruptly terminates the connection (line 8), the server receives the disconnect (line 9), and both users disable and release their respective endpoints (lines 10 and 11).

When using the orderly release feature, users typically call the same sequence of functions until it is time to terminate the connection, as shown in Table 7.

Table 7 Function calls for connection-mode service with orderly release

Client	Server
1. t_open()	t_open()
2. t_bind()	t_bind()
3.	t_listen()
4. t_connect()	
5.	t_accept()
6. t_snd()	
7.	t_rcv()
8. t_sndrel()	
9.	t_rcvrel()
10.	t_sndrel()
11. t_rcvrel()	
12. t_unbind()	t_unbind()
13. t_close()	t_close()

To end the session, the client sends an orderly release request indicating it has no more data to send (line 8). The server receives it and then sends its own orderly release request (lines 9 and 10). When the client receives the orderly release request (line 11), the transport provider terminates the connection. An empty space in a column for one of the users indicates a function call by the other user prior to the specified user requesting the next function.

Orderly release indicates that a user has finished sending data and informs the transport provider that the user is cooperating in the connection termination. Both users must send an orderly release and respond to the peer orderly release before the connection is terminated.

The following sections describe each phase of connection-mode service and the related TLI library functions.

Initializing and deinitializing an endpoint

The functions that the TLI library uses to initialize and deinitialize an endpoint are:

- `t_open()`

  Creates a transport endpoint to a specified transport provider and allocates internal data structures and buffers. This function returns a local file descriptor (`fd`) for the end point and default protocol information, such as the maximum length of the Transport Service Data Unit (TSDU) or Expedited Transport Service Data Unit (ETSDU).

- `t_bind()`

  Associates a protocol address with an opened endpoint. For a listening application using connection-mode service, set the `qlen` field to a number greater than zero. This setting enables the transport provider to queue multiple connection requests for the endpoint. Incoming connection requests can be accepted on another endpoint; for the other endpoint, set `qlen` to zero.

- `t_optmgmt()`

  Obtains the protocol options for a bound endpoint. The transport provider either returns the supported options or allows the user to negotiate for them.

- `t_unbind()`

  Disables an endpoint and flushes any read and write data queues. Use this function once data transfer operations over the endpoint are complete and established connections are released. Once an endpoint is disabled, the transport provider rejects subsequent data and events for the specified endpoint. At this time, you can call `t_bind()` to associate a new protocol address to the endpoint.

- `t_close()`

  Releases an endpoint, the local file descriptor, and associated local resources.

These functions affect only the local endpoint and do not result in transfer of information across the network.

Establishing a connection

To establish a connection between two peer transport users, the users must first initialize endpoints in their local environment. For each end of the connection, the endpoint must be opened and bound to a protocol address. Refer to the section "Initializing and deinitializing an endpoint" on page 164 for information on how to initialize an endpoint for various types of applications.

When you use the same endpoint to receive and accept a connect request, subsequent connection requests to the address bound to that endpoint are rejected by the transport provider until the current connection is closed.

The TLI library functions used to establish a transport connection are:

- `t_connect()`

  Connects to a transport user and, if included, sends user data to the server. In synchronous-mode execution, control is returned to the client after the transport user responds. Control is returned immediately in asynchronous mode.

- `t_rcvconnect()`

  Receives connect confirmation and, if available, returns user data from the transport user. This function applies only to asynchronous mode. A connection is established after successful return of this function.

- `t_listen()`

  Listens for a connection request. If an outstanding request exists, return any user data and the unique connection indicator.

- `t_accept()`

  Accepts a connection request and, if included, sends user data to the client.

Transferring data

A transport connection between two users provides a full-duplex data transfer path across the network. The TLI library includes two data transfer functions for connection-mode service. These functions are:

- `t_snd()`

  Sends normal or expedited data over a connection.

- `t_rcv()`

  Receives normal or expedited data from a connection.

Data transfer operations are affected by choice of execution mode, occurrence of an asynchronous event, and use of expedited data.

Receiving data

The `t_rcv()` function returns available data. If the transport endpoint specified in the call does not correspond to an established connection, the function returns a `TBADF` error.

If data is not available, the return value depends on the execution mode, as follows:

- Asynchronous mode

  Function returns a `TNODATA` error. Use additional `t_rcv()` or `t_look()` calls to poll for data.

- Synchronous mode

  Function call is blocked until the transport provider detects:
  - Incoming normal or expedited data. The function returns the data.
  - Disconnect event. The function returns -1and sets `t_errno` to `TLOOK`. Call `t_look()` to retrieve the event.
  - Orderly release event. The function returns -1and sets `t_errno` to `TLOOK`. Call `t_look()` to retrieve the event.

Incoming data may be fragmented if the transport provider cannot pass a complete TSDU or ETSDU to the user. Expedited data can interrupt normal data. To monitor for these conditions, check the bit field in the flag set by `t_rcv()`:

- `T_EXPEDITED`

  Set for expedited data. Clear for normal data.

- `T_MORE`

  Set if a complete TSDU or ETSDU was not received; use additional `t_rcv()` calls to receive the remaining fragments. Clear if last fragment or if complete TSDU or ETSDU was received.

Sending data

Call the t_snd() function to send normal or expedited data. If the data length exceeds the maximum length of the TSDU or ETSDU, use multiple t_snd() calls to send the data. Set the T_MORE flag as follows:

- Set

  Additional TSDU or ETSDU fragments to follow.

- Clear

  Last fragment in the TSDU or ETSDU.

The transport provider accepts the data and returns the length of data accepted in bytes. Flow control may prevent the provider from accepting all or part of the data. The effect on the t_snd() call depends on the execution mode, as follows:

- Asynchronous mode

  Returns an error if no data could be accepted or the returned nbytes is less than the request. Call t_snd() to issue one or more calls to send the remainder of the data.

- Synchronous mode

  Function is blocked until flow control is cleared and the transport provider can accept all of the data.

Regardless of execution mode, normal and expedited data are two distinct flows. Set the T_EXPEDITED flag to indicate expedited data. You can send expedited data even if flow control is in effect for normal data.

Two asynchronous events, T_DISCONNECT and T_ORDREL, may occur during execution of t_snd(), changing the state of the interface. The effect of these events on the t_snd() function call is as follows:

- Disconnect

  Function returns -1 and sets t_errno to TLOOK. If an asynchronous mode function call has already returned, you must poll for asynchronous events using t_look().

- Orderly release

  Function returns -1 and sets t_errno to TLOOK. A user can still send data after receiving an orderly release indication but can no longer receive data.

Terminating a connection

The TLI library supports two types of connection terminations: abortive and orderly. The two types differ in the preservation of user data.

- Disconnect indication

  A disconnect indication (abortive release) may be issued by either user of a connection during connection establishment or data transfer. The release takes immediate effect without negotiation between users; queued data is lost. This type of termination results in a `T_DISCONNECT` asynchronous event for the peer user. If a transport provider initiates an abrupt termination, both users are notified of the termination.

- Orderly release

  In an *orderly release*, cooperating users can release a connection without data loss. Either user can send an orderly release request indicating that outgoing data transfer is complete. The peer user responds, sends any remaining data, and then also sends an orderly release request. Once the initiating user responds, the connection is released. Orderly release is simply a way for two users to inform the respective transport providers that data transfer in a given direction is complete and the connection is no longer needed.

As an alternative to abortive or orderly release, cooperating peer users can implement a higher-level mechanism to notify the remote user of an upcoming connection release and prevent data loss. For example, a user may send a specific expedited message to a remote user and wait for the response before aborting the connection.

The TLI library functions that support connection release are:

- `t_snddis()`

  Releases a transport connection and sends user data to the peer user. While a connection is being established, use this function to reject the connection request and send a `sequence` value indicating the rejected connection indication.

- `t_rcvdis()`

  Acknowledges an abrupt termination and retrieves user data, a `sequence` value, and a reason, if provided.

- `t_sndrel()`

Initiates the orderly release of a connection. Subsequent `t_snd()` calls are rejected.

- `t_rcvrel()`

  Responds to an orderly release.

Connectionless-mode functions

Connectionless-mode service involves two phases:

- Initializing and deinitializing an endpoint
- Transferring data across the endpoint

To use connectionless-mode service, a user process creates a transport endpoint and associates a protocol address with the endpoint. The endpoint can then be used to send datagrams (connectionless-mode data) to a peer user.

When the data transfer is complete, the user disables the endpoint to free local resources or binds a new protocol address to the endpoint.

Table 8 lists a typical sequence of function calls used for connectionless-mode service.

Table 8 Function calls for connectionless-mode service

Active user	Passive user
1. `t_open()`	`t_open()`
2. `t_bind()`	`t_bind()`
3. `t_sndudata()`	
4.	`t_rcvudata()`
5. `t_unbind()`	`t_unbind()`
6. `t_close()`	`t_close()`

This table does not include all the functions a user might call; rather, only those critical to connectionless-mode service. An empty space in a column for either user indicates a function call made by the other user prior to the specified user requesting the next function. Both users initialize their respective endpoints (lines 1 and 2), exchange data units (lines 3 and 4), and then deinitialize the endpoints (lines 5 and 6). Depending on the application, other functions may also be called.

The following subsections describe each phase of connectionless-mode service and the related library functions.

Initializing and deinitializing an endpoint

The functions used to initialize and deinitialize an endpoint include:

- `t_open()`

  Creates a transport endpoint to a specified transport provider and allocates internal data structures and buffers. This function returns default protocol information, such as the maximum length of the TSDU and the local file descriptor (`fd`) for the endpoint.

- `t_bind()`

  Associates a protocol address with an opened endpoint. For connectionless-mode service, the `qlen` field is not used. Set the `req` parameter to a null pointer to have the transport provider assign an address.

- `t_setpeer()`

  Creates a default destination address. This causes all data sent through this endpoint to be sent to the specified address. Only datagrams originating from the remote endpoint will be received.

- `t_optmgmt()`

  Obtains or negotiates the protocol options for a bound endpoint.

- `t_unbind()`

  Disables an endpoint and flushes any read and write data queues. Use this function once data transfer operations over the endpoint are complete. Once an endpoint is disabled, the transport provider rejects subsequent data and events for the specified endpoint. At this time, you can call `t_bind()` to associate a new protocol address with the endpoint.

- `t_close()`

  Releases an endpoint, the local file descriptor, and associated local resources.

These functions affect only the local endpoint and do not result in the transfer of information across the network.

Transferring data

Users can transfer connectionless mode data through an initialized endpoint. The units are self-contained messages since connectionless operation does not sequence data. This service supports normal (nonexpedited) data only.

The TLI library supports the following data transfer functions for connectionless-mode service:

- `t_sndudata()`

  Sends unit of data to the specified address.

- `t_rcvudata()`

  Receives unit of data.

- `t_rcvuderr()`

  Retrieves information about an error in a previous data unit.

- `t_snd`
  `t_rcv`

  These functions are supported on datagram connections that have done a `t_setpeer`. Otherwise, an error is returned.

During data transfer, an asynchronous event may occur that affects the connection or state of the transport provider. The most common event in this phase is `T_DISCONNECT`. If an asynchronous event occurs, `t_sndudata()` or `t_rcvudata()` returns -1 and sets `t_errno` to `TLOOK`. Use the `t_look()` function to retrieve the event.

Receiving data

If data is not available, the function call return value depends on the execution mode, as follows:

- Asynchronous mode

  Function returns a `TNODATA` error. Use additional `t_rcvudata()` or `t_look()` calls to poll for data.

- Synchronous mode

  Function call is blocked until the transport provider receives data.

If an asynchronous event occurs, the function returns -1 and sets `t_errno` to `TLOOK`. Use the `t_look()` function to retrieve the event.

If the transport provider receives an error indication for a previously transmitted data unit, the t_rcvudata() function returns -1 and sets t_errno to TLOOK. The event type returned by t_look() is T_UDERR. Call t_rcvuderr() to retrieve error information.

Incoming data may be fragmented if the transport provider cannot pass a complete TSDU to the user. To monitor for this condition, check the flags field returned by t_rcvudata(). If T_MORE is set, the complete TSDU was not passed to the user. Use additional t_rcvudata() calls to retrieve the remaining data.

Sending data

The t_sndudata() function enables you to send a unit of data. The data length cannot exceed the maximum length of a TSDU.

The transport provider accepts the data unit immediately if able and returns 0 on successful completion. Flow control may prevent the provider from accepting data. The effect of flow control on the t_sndudata() call depends on the execution mode, as follows:

- Asynchronous mode

  Returns an error and sets t_errno() to TFLOW. The process must re-issue the t_sndudata() call at a later time.

- Synchronous mode

  Function is blocked until flow control is cleared and transport provider can accept all of the data.

t_sndudata() does not detect asynchronous events. User processes executing in synchronous or asynchronous mode must poll for asynchronous events using t_look().

If the destination endpoint does not exist, has not been initialized by a remote user, or has been deinitialized, the data unit may be discarded.

Utility functions

The TLI library provides additional utility functions to control an endpoint and manage memory. Functions in this group can be called at any time and do not result in the transfer of information over the network.

These functions include:

- t_getinfo()

  Returns protocol information for the specified endpoint.

- t_getstate()

Returns the current state of the transport provider associated with the specified transport endpoint.

- `t_sync()`

  Synchronizes library data structures (maintained for the specified transport endpoint) with the underlying transport provider.

- `t_alloc()`

  Allocates and initializes memory space for the specified TLI library data structure.

- `t_free()`

  Frees memory space provided by the `t_alloc()` function.

- `t_error()`

  Prints a user-specified message along with a standard error message corresponding to the current value of `t_errno`.

- `t_look()`

  Returns the current event for the specified transport endpoint. If no event is found, zero is returned.

Events and states 10

Overview

This chapter provides information on:

- Events that occur as the result of activity on an endpoint
- Issues concerning the state of the transport interface as viewed by a user process, including:
  - Rules a user process must follow to maintain the state of the transport interface
  - State transition tables
  - State transition diagrams for connection-mode and connectionless-mode service

Throughout this chapter, event and state descriptions reference the following symbolic constants:

- `T_COTS`—Connection-mode service
- `T_COTS_ORD`—Connection-mode service, with orderly release
- `T_CLTS`—Connectionless-mode service

The following support functions do not affect the state, can be called at any time after initializing an endpoint, and are excluded from state rules:

- `t_getstate()`
- `t_getinfo()`
- `t_alloc()`
- `t_free()`
- `t_look()`
- `t_error()`
- `t_sync()`

Outgoing events

When you invoke a TLI library routine that requests a service or responds to an event, the routine sends an appropriate primitive to the transport provider. Outgoing events are defined as the successful return of library functions that send a request to the transport provider or which respond to an event previously received from the transport provider.

The meaning of some outgoing events depends on the context in which they occur. For example, three different `t_accept()` events are possible, each distinguished by a different context. The context is defined by the contents of the following variables:

- `ocnt`

  The number of outstanding connect indications that the user process has neither rejected nor accepted.

- `fd`

  The local file descriptor for the listening endpoint; that is, the endpoint used to receive connect indications.

- `resfd`

  The file descriptor associated with the transport endpoint to which a connection will be transferred using the `t_accept()` library routine.

The use of the variable `ocnt` by an application program should not be taken literally. A program should keep track of how many connection indications are outstanding on a given transport endpoint, but may do so implicitly or explicitly. In addition, variable names other than `ocnt`, `fd`, and `resfd` may be used.

Table 9 describes each type of outgoing event and lists the service modes in which the event can occur.

Table 9 Outgoing events

Event	Description	Service mode
opened	t_open() returned successfully.	T_COTS, T_CLTS, T_COTS_ORD
bind	t_bind() returned successfully.	T_COTS, T_CLTS, T_COTS_ORD
optmgmt	t_optmgmt() returned successfully.	T_COTS, T_CLTS, T_COTS_ORD
unbind	t_unbind() returned successfully.	T_COTS, T_CLTS, T_COTS_ORD
closed	t_close() returned successfully.	T_COTS, T_CLTS, T_COTS_ORD
connect1	t_connect() synchronous mode returned successfully.	T_COTS, T_COTS_ORD
connect2	t_connect() asynchronous mode returned TNODATA error, or disconnect indication arrived at the endpoint resulting in a TLOOK error.	T_COTS, T_COTS_ORD
accept1	t_accept() returned successfully; ocnt==1, fd==resfd.	T_COTS, T_COTS_ORD
accept2	t_accept() returned successfully; ocnt==1, fd!=resfd.	T_COTS, T_COTS_ORD
accept3	t_accept() returned successfully; ocnt>1.[1]	T_COTS, T_COTS_ORD
snd	t_snd() returned successfully.	T_COTS, T_COTS_ORD
snddis1	t_snddis() returned successfully; ocnt<=1.[1]	T_COTS
snddis2	t_snddis() returned successfully; ocnt>[1].	T_COTS
sndrel	t_sndrel() returned successfully.	T_COTS_ORD
sndudata	t_sndudata() returned successfully.	T_CLTS
setpeer	t_setpeer() returned successfully.	T_CLTS

1. onct is significant only to a listening endpoint.

Incoming events

Incoming events are defined as the successful return of library functions that receive data or event information from the transport provider.

The meaning of some incoming events depends on the context in which they occur. For example, three different `t_rcvdis()` events are possible, each distinguished by a different context. The context is defined by the contents of `ocnt`: the number of outstanding connection indications (connections that the user process has neither rejected nor accepted on a given transport endpoint).

Table 10 describes each incoming event and lists the service modes in which the event can occur.

Table 10 Incoming events

Event	Description	Service mode
`listen`	`t_listen()` returned successfully.	`T_COTS, T_COTS_ORD`
`rcvconnect`	`t_rcvconnect()` returned successfully.	`T_COTS, T_COTS_ORD`
`rcv`	`t_rcv()` returned successfully.	`T_COTS, T_COTS_ORD`
`rcvdis1`	`t_rcvdis()` returned successfully; `ocnt==0`.	`T_COTS`
`rcvdis2`	`t_rcvdis()` returned successfully; `ocnt==1`.	`T_COTS`
`rcvdis3`	`t_rcvdis()` returned successfully; `ocnt>1`.	`T_COTS`
`rcvrel`	`t_rcvrel()` returned successfully.	`T_COTS_ORD`
`rcvudata`	`t_rcvudata()` returned successfully.	`T_CTLS`
`rcvuderr`	`t_rcvuderr()` returned successfully.	`T_CTLS`
`pass_conn`	Received a transferred connection.	`T_COTS, T_COTS_ORD`

In general, incoming events are associated with the endpoint on which the event notification arrived. One exception is the `pass_conn` event. By definition, this event occurs when `t_accept` is used to establish a connection on a different endpoint than the one on which the connect indication arrived. Refer to the t_accept(3) man page for details.

Asynchronous events

Incoming events are defined as "asynchronous" if they are retrieved by a library routine that expects some other event. For example, `t_rcv()` expects to retrieve data. If it retrieves a disconnect indication instead, the event is considered asynchronous. The library routine returns `-1` and sets `t_errno` to `TLOOK` when it encounters an asynchronous event. It is up to the application program to invoke the `t_look()` function to determine which event occurred and invoke the appropriate routine for that event.

Continuing with our example, suppose `t_look()` returns `T_DISCONNECT`. The application should then invoke `t_rcvdis()` to handle the disconnect indication.

Note

Do not confuse the use of the term "asynchronous" to describe events with its use to describe execution mode. An application program may encounter asynchronous events when executing in either synchronous or asynchronous mode.

Synchronous and asynchronous execution modes are described in the section "Execution modes" on page 156.

Asynchronous events can occur in either execution mode and are shown in Table 11.

Table 11 Asynchronous events

Symbolic name	Event description
`T_LISTEN`	Connection indication received. Connection-mode service only. Occurs if: (1) file descriptor is bound to a valid protocol address with `qlen` parameter > 0, and (2) connection is not in effect on the endpoint.
`T_CONNECT`	Connection response received after return of `t_connect()` function. Connection-mode service only.
`T_DATA`	Normal data received.
`T_EXDATA`	Expedited data received.
`T_DISCONNECT`	Disconnect indication received after return of `t_acccept()`, `t_snddis()`, or any data transfer function. Connection-mode service only.

Table 11 Asynchronous events (continued)

Symbolic name	Event description
T_ORDREL	Orderly connection release indication received. Connection-mode service only.
T_UDERR	Error detected in last datagram sent by local user. Reported on return from t_rcvudata() or t_unbind() function. Connectionless-mode service only.

Table 12 associates asynchronous events with TLI functions. Note that not all TLI functions set t_errno to TLOOK.

Table 12 Functions that return TLOOK errors

Function	TLOOK Error Cause
t_accept()	T_DISCONNECT or T_LISTEN event occurred.
t_bind()	Unexpected incoming event.
t_connect()	T_DISCONNECT or T_LISTEN event occurred. The T_LISTEN event occurs only when the function is called on a listening endpoint (t_bind() function with qlen>0) and a connection is pending.
t_getinfo()	Unexpected incoming event.
t_getstate()	Unexpected incoming event.
t_listen()	T_DISCONNECT event occurred; user process has not accepted or rejected an outstanding connection indication which it received earlier on the endpoint.
t_optmgmt()	Unexpected incoming event.
t_rcv()	T_DISCONNECT or T_ORDREL event occurred.
t_rcvconnect()	T_DISCONNECT event occurred.
t_rcvrel()	T_DISCONNECT event occurred.

Table 12 Functions that return TLOOK errors (continued)

Function	TLOOK Error Cause
t_rcvudata()	T_UDERR event.
t_rcvuderr()	Unexpected incoming event
t_sync()	Unexpected incoming event.
t_unbind()	T_LISTEN event occurred.

Once an asynchronous event occurs on an endpoint, subsequent calls to the same function or to other functions that are affected by the same event continue to return a TLOOK error. The error condition remains in effect until a user process clears the event with the appropriate TLI routine; these processes are referred to as *consuming* functions.

Table 13 lists each asynchronous event and the corresponding consuming function.

Table 13 Asynchronous event clearing mechanism

Event	Consuming function
T_LISTEN	t_listen()
T_CONNECT	t_rcvconnect()
T_DATA	t_rcv(), t_rcvudata()
T_EXDATA	t_rcv()
T_DISCONNECT	t_rcvdis()
T_ORDREL	t_rcvrel()
T_UDERR	t_rcvuderr()

States

The TLI library uses a state-driven interface. In general, calls to library functions that do not violate the current state of the interface and transport provider are accepted; others are rejected.

For a given transport endpoint, TLI appears to the user to be in one of eight possible states at any given time. Several of the states apply only to connection-mode data transfer. Two states, `T_OUTREL` and `T_INREL`, are valid only if orderly release is being used on the connection.

Table 14 lists TLI states, descriptions, and their service types.

Table 14 TLI states

State	Description	Service type
`T_UNINIT`	Uninitialized state. First and last state of the transport interface.	`T_COTS, T_CLTS, T_COTS_ORD`
`T_UNBND`	User has not bound a protocol address to the endpoint.	`T_COTS, T_CLTS, T_COTS_ORD`
`T_IDLE`	An address has been bound to the transport endpoint.	`T_COTS, T_CLTS, T_COTS_ORD`
`T_OUTCON`	User has sent a connect request (outgoing event) but has not yet received a response.	`T_COTS, T_COTS_ORD`
`T_INCON`	User has received a connect indication (incoming event) which it has not yet accepted or rejected.	`T_COTS, T_COTS_ORD`
`T_DATAXFER`	Endpoint is in data transfer state.	`T_COTS, T_COTS_ORD`
`T_OUTREL`	Local user issued orderly release request. Waiting for orderly release indication from remote user.	`T_COTS_ORD`
`T_INREL`	An orderly release request has been received from a remote user, but has not been sent by the local user.	`T_COTS_ORD`

Rules for maintaining states

Transport providers enforce the following rules to maintain the appropriate state of the interface:

- The transport provider maintains a record of the interface state.
- The transport provider rejects functions that would violate the current state of the interface, setting `t_errno` to `TOUSTATE`.
- If a user process calls functions out of order, the transport provider maintains the current state and returns an error indication. The transport provider rejects any data provided with the function.
- The initial state of an endpoint is `T_UNINIT`. A user process must first initialize the endpoint and associate it with a protocol address before the transport provider views the endpoint as active.
- The final state of an endpoint is also `T_UNINIT`. While in this state, the endpoint cannot be used.
- The `t_close()` routine is normally called from the `T_UNBND` state. However, it may be called from any state to abort the indicated transport connection.

For connection-mode service, transport providers also enforce the following rules:

- A user can initiate a connection release at any time while establishing a connection or transferring data.
- The `t_accept()` function is the only way of transferring the interface state of one endpoint to another endpoint. The destination endpoint must be bound to a protocol address and must be in the idle (`T_IDLE`) state. If there are no other outstanding connection indications for the source endpoint, the transport provider places the source endpoint in the idle `T_IDLE` state when the transfer is complete. Otherwise, the source endpoint remains in the `T_INCON` state.

State transition tables

A state transition is the process of moving from one state to the next based on the occurrence of a specific event.

Table 15, Table 16, and Table 17 illustrate the state transitions of the transport interface as viewed by the user. The layout of these tables is as follows:

- Column headings indicate the current state
- Row headings indicate an event
- Column-row intersections indicate the next state. An empty intersection indicates an invalid event for the given state.

During certain transitions, the transport user must take specific actions. The notation [*n*] used in the state transition tables represents a required user action, as follows:

[1]— Set to zero the count of outstanding connection indications

[2]— Increment the count of outstanding connection indications

[3]— Decrement the count of outstanding connection indications

[4]— Transfer a connection to another endpoint with the `t_accept()` routine

As mentioned earlier, these user activities should not be taken too literally. Refer to "Outgoing events" on page 176.

Table 15 contains a state transition table for initializing and de-initializing an endpoint.

Table 15 State transition table (part 1)

Event	T_UNINIT	T_UNBIND	T_IDLE
opened	T_UNBIND		
bind		T_IDLE[1]	
optmgmt			T_IDLE
unbind			T_UNBIND
closed		T_UNINIT	T_UNINIT

Table 16 contains a state transition table for connectionless-mode data.

Table 16 State transition table (part 2)

Event	T_IDLE
sndudata	T_IDLE
rcvudata	T_IDLE
rcvuderr	T_IDLE

Table 17 contains state transition table for establishing a connection, connection-mode data transfer, and terminating a connection.

Table 17 State transition table (part 3)

Event	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER	T_OUTREL	T_INREL
connect1	T_DATAXFER					
connect2	T_OUTCON					
rcvconnect		T_DATAXFER				
listen	T_INCON[2]		T_INCON[2]			
accept1			T_DATAXFER [3]			
accept2			T_IDLE [3,4]			
accept3			T_INCON [3,4]			
snd				T_DATAXFER		T_INREL
rcv				T_DATAXFER	T_OUTREL	
snddis1		T_IDLE	T_IDLE[3]	T_IDLE	T_IDLE	T_IDLE
snddis2			T_INCON[3]			
rcvdis1		T_IDLE		T_IDLE	T_IDLE	T_IDLE
rcvdis2			T_IDLE[3]			
rcvdis3			T_INCON[3]			
sndrel				T_OUTREL		T_IDLE
rcvrel				T_INREL	T_IDLE	
pass_con	T_DATAXFER					
closed	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT

State diagrams

Figure 57 through Figure 59 illustrate the flow between states for connection-mode (Figure 57), connection-mode with orderly release (Figure 58), and connectionless-mode service (Figure 59).

Note that these diagrams do not show every possible sequence in which the TLI routines may be called. For example, in Figure 57 and Figure 58, the client is assumed to be executing in asynchronous mode. It therefore invokes `t_rcvconnect` after invoking `t_connect`. If it were executing in synchronous mode, `t_rcvconnect` would not be invoked. Upon returning from `t_connect`, the endpoint would be in the `T_DATAXFER` state.

Figure 57 Connection-mode service state diagram

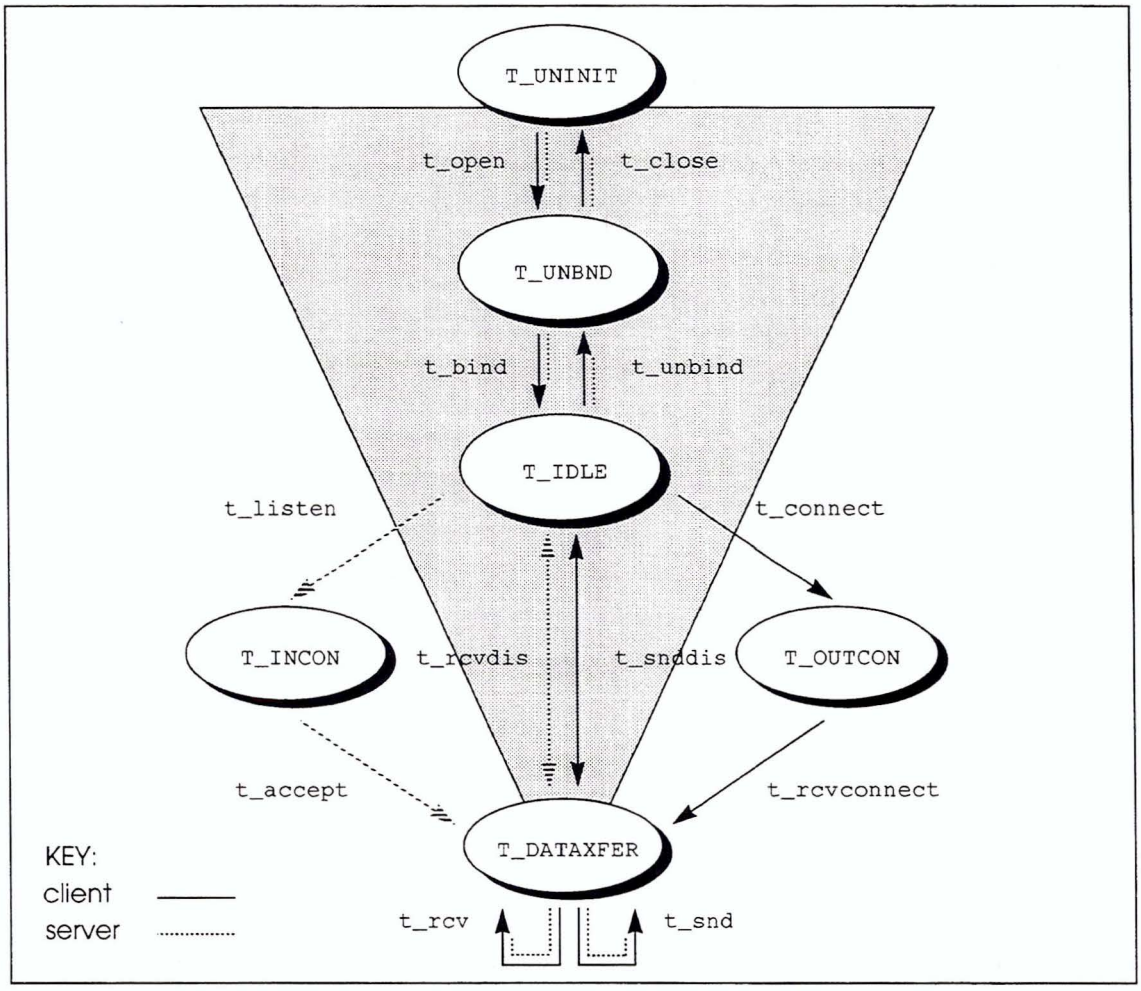
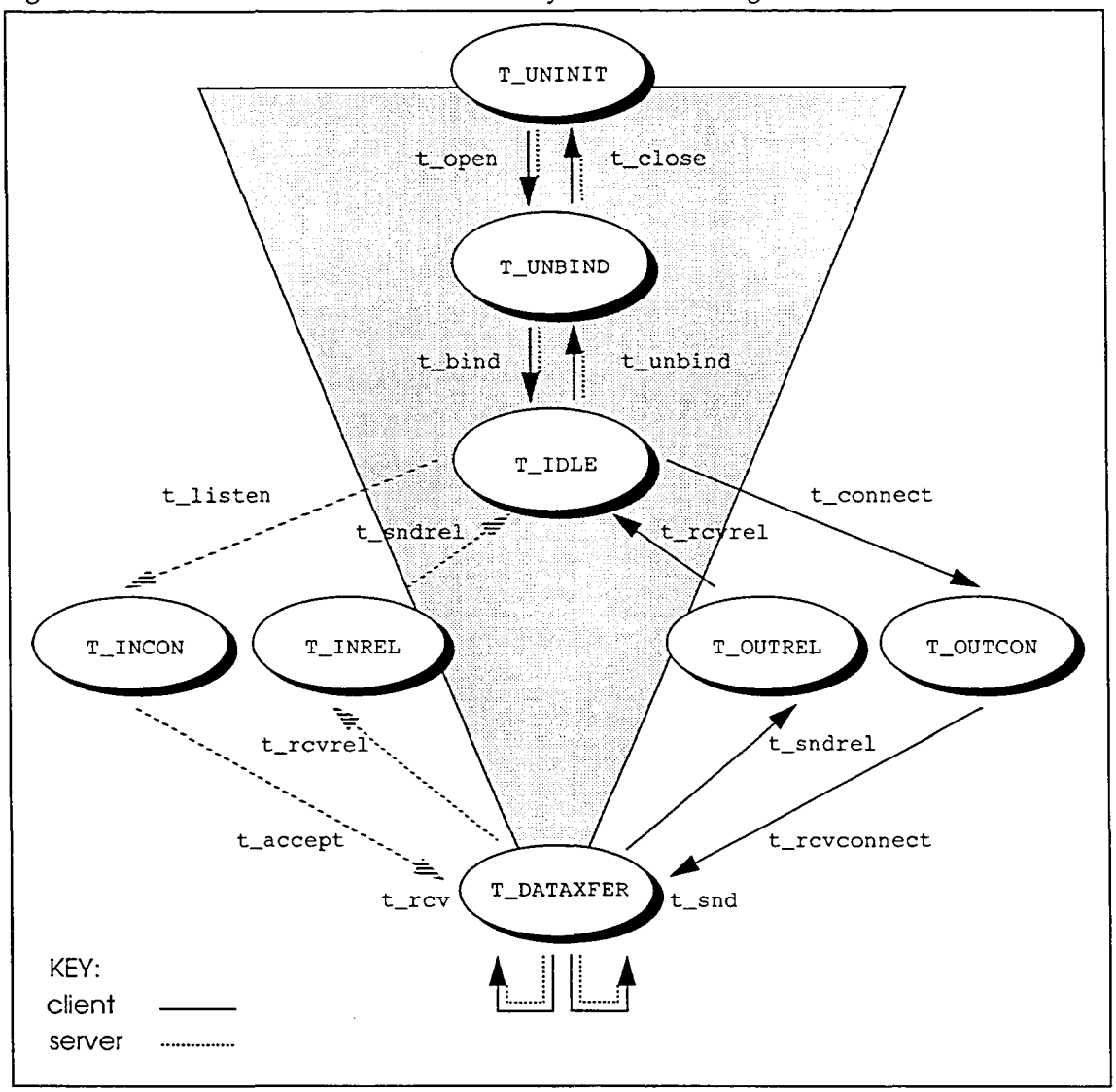
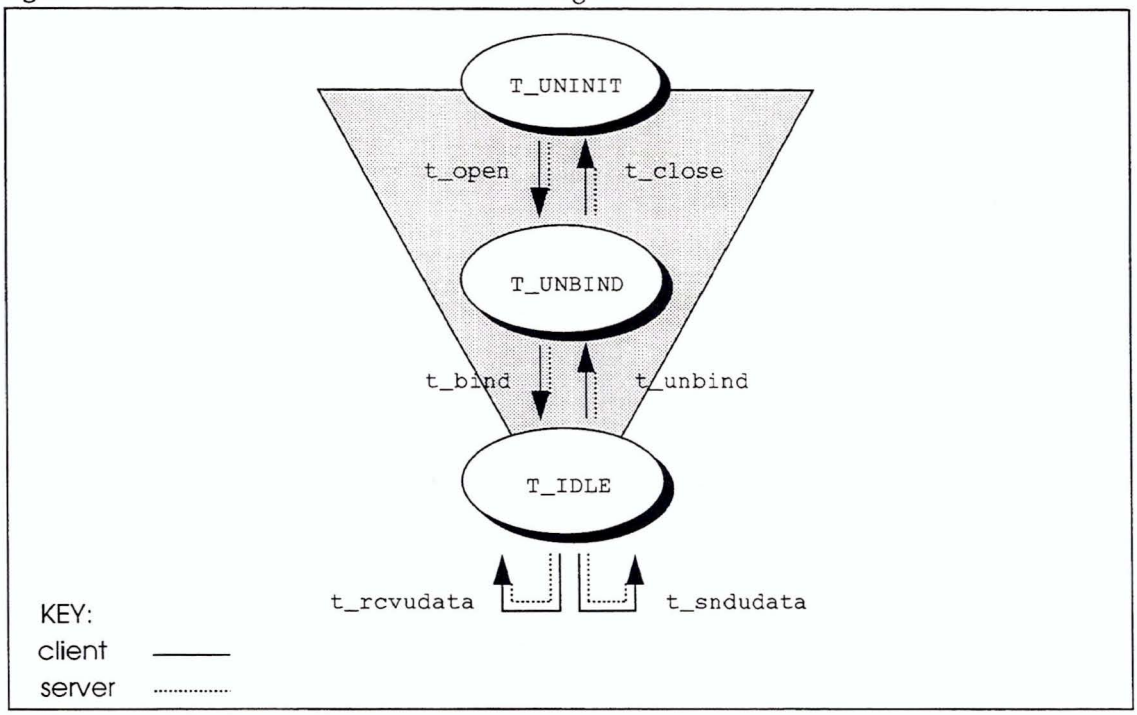


Figure 58 Connection-mode service with orderly release state diagram



In Figure 57 and Figure 58, it is assumed that the client will initiate the disconnect.

Figure 59 Connectionless-mode service state diagram



TLI programming example 11

Overview

The TLI library is bundled with ConvexOS and consists of the following files:

- `libnsl.a`—TLI library archive
- `tiuser.h`—Header file that must be included in applications that use the TLI library

This chapter provides information on compiling user applications that call TLI library functions. A sample user program is included to illustrate a subset of the functions.

Compiling the user application

The TLI library is single-threaded. Do not specify optimization level 3 when compiling a user program that uses the library. A sample compilation is invoked as follows:

```
cc -o appl appl.c -lnsl
```

In this sample compilation, the user application source file name is `appl.c` and the compiled output is `appl`. The `-l` switch identifies which library to use when linking the application program. The compiler prepends "lib" and appends ".a" to "nsl" to form "libnsl.a."

Sample user program

The program shown in Figure 60 is an example of a passive application that listens for incoming connection requests, establishes connections in asynchronous mode, and receives data concurrently on each of the connections.

Note

This example is not complete—many details have been omitted for the sake of simplicity. For example, not all data structures are declared or initialized, and there is insufficient error checking.

Figure 60 Example server programming application

```
                                /* SERVER PROGRAM */
#include <tiuser.h>
extern int t_errno;

int     fd[MAX_CONNEC]; /* array of file descriptors */
int     lis_fd;         /* file descriptor of listening endpoint */
int     i;

main()
{
        /* initialize fd array */
        for (i=0; i < MAX_CONNEC; i++)
                fd[i] == -1;
        /* open and bind listening endpoint */
        lis_fd = fd[0] = t_open("/dev/cox", O_RDWR | O_NONBLOCK, p_info);
        t_bind(lis_fd, req, ret);
        for (;;)
        {
                for (i=0; i < MAX_CONNEC; i++)
                {
                        if ( fd[i] == -1 )
                                continue;
                        switch(t_look(fd[i]))
                        {
                                case T_LISTEN:
                                    handle_connec();
                                case T_DISCONNECT:
                                    handle_discon(i);
                                case T_DATA:
                                case T_EXDATA:
                                    handle_data(fd[i]);
                                case T_ERROR:
                                    fprintf(stderr, "t_look returned error event");
                                    exit(1);
                                case 0: /* no event */
                                    continue
                                case -1:
                                    fprintf(stderr, "t_look failed, t_errno: %d",
                                     t_errno);
                                    exit(2);
                                default:
                                    fprintf(stderr, "t_look returned unexpected
                                     event");
                                    exit(3);
                        }
                }
        }
}
```

Figure 60 Sample server programming application (continued)

```
                    /* SERVER PROGRAM CONT. */
handle_connec() /* handle incoming connect indication */
{
        t_listen(lis_fd, pcall); /* retrieve connect indication */
        index = find_vacant(); /* find vacant fd array entry */
        if (index == -1)
        {
                /* refuse connect indication, since fd array is full */
                t_snddis(lis_fd, pcall);
        }
        else
        {
                fd[index] = t_open("/dev/cox", O_RDWR|_NONBLOCK, NULL);

                t_bind(fd[index], NULL, NULL);

                /* establish connection on different endpoint from
                                that used for listening */

                t_accept(lis_fd, fd[index], pcall);
        }
}

handle_data(fdes) /* handle incoming data */
int fdes;
{
        char   buf[500];
        int    flags;

        t_rcv(fdes, buf, sizeof(buf), &flags);
                .
                .
                .

        /* code to process incoming data omitted */
}
handle_discon(index) /* handle disconnect indication */
int index;
{
        /* retrieve disconnect indication */
        t_rcvdis(fd[index], &discon);
        /* unbind and close endpoint */
        t_unbind(fd[index]);

        t_close(fd[index]);

        fd[index] = -1;
}
```

Note that the O_NONBLOCK flag is specified in the t_open() calls; this selects asynchronous execution mode for the given endpoint. The use of asynchronous mode facilitates concurrent processing of several transport connections. Lack of activity in one endpoint does not prevent processing of activity on other endpoints.

/dev/cox is assumed to be the name of a connection-oriented transport provider.

Standards compliance of the CONVEX TLI

12

The CONVEX Transport Layer Interface (TLI) provides a consistent transport-level interface that allows user applications to access any CONVEX transport provider. Because the interface is based on international standards, user applications are assured of specific provider independence and are easily ported between systems.

The interface provides Open Systems Interconnection (OSI) transport-level services and conforms to the following standard interface definitions:

- X/Open Transport Interface Issue 3 (XTI 3)
- System V Interface Definition Issue 4 (SVID 4)

XTI is based on SVID and defines the same library functions to access the transport interface. However, XTI 3 includes several enhancements:

- More routines potentially set `t_errno()` to `TOUTSTATE`, which indicates an out-of-sequence function call.
- Additional functions return `TLOOK` error indicating the occurrence of an asynchronous event.

The CONVEX Transport Layer Interface library implements the same functions as XTI and SVID, with the few minor differences described in the following sections.

TLOOK errors

Table 18 shows which TLI routines (potentially) set t_errno() to TLOOK in the CONVEX implementation and under the SVID and X/Open implementations.

Table 18 Comparison of functions that return a TLOOK error

Function	CONVEX TLI	SVID	XTI
t_accept()	x	x	x
t_alloc()			
t_bind()	x		
t_close()			
t_connect()	x	x	x
t_error()			
t_free()			
t_getinfo()	x		
t_getstate()	x		
t_listen()	x	x	x
t_look()			
t_open()			
t_optmgmt()	x		
t_rcv()	x	x	x
t_rcvconnect()	x	x	x
t_rcvdis()			
t_rcvrel()	x	x	x
t_rcvudata()	x	x	x
t_rcvuderr()			
t_snd()	x		x
t_snddis()	x	x	
t_sndrel()			x

Table 18 Comparison of functions that return a TLOOK error (continued)

Function	CONVEX TLI	SVID	XTI
t_sndudata()			x
t_sync()	x		
t_unbind()	x	x	x

Error codes

The CONVEX TLI library includes additional error codes to detect additional error conditions in the transport providers, the library, or user processes. The constants for these error codes are:

- TBADPARAM—The user passed an invalid argument to the TLI library function.
- TWRONGACK—The transport provider returned to the library routine an acknowledgment to a primitive other than the last one sent.
- TIMPASYEV—The transport provider returned an asynchronous event notification that is not valid for the called TLI library function.

t_alloc differences

The SVID states that the t_alloc() routine should fail (return -1 and set t_errno() to TSYSERR) if the size value associated with any specified field is -1 or -2. The size value being referred to here is any of the buffer-size values returned in the T_info_ack structure by the t_open() and t_getinfo() routines.

However, some applications depend on t_alloc **not** failing when -1 has been returned by the transport provider for the buffer size, and instead expect t_alloc to use some default for the buffer size.

To accommodate such applications, the CONVEX implementation of t_alloc() uses default buffer sizes of 64, 256, and 2048 for the T_ADDR, T_OPT, and T_UDATA buffers, respectively, if the size value for any of these buffers is -1.

For -2, t_alloc() still fails.

Preventing indefinite blocking

Using the SVID definition, a user process could be blocked indefinitely if certain functions are invoked inappropriately. The conditions in which indefinite blocking could occur include:

- The user calls `t_rcv()` after retrieving an orderly release indication (`T_ORDREL`).
- The user calls `t_listen()` on an endpoint that was not previously bound for listening (in other words, with the `qlen` parameter set to zero in the `t_bind()` function call for the endpoint).

To prevent indefinite blocking of a user process, both XTI and CONVEX TLI implement additional error checking to the above-mentioned functions.

Asynchronous events

CONVEX implements all of the asynchronous events defined by SVID. XTI adds two additional events, `T_GODATA` and `T_GOEXDATA`, which CONVEX TLI does not presently support.

Option information buffer format

13

This chapter describes the option information buffer format for the Internet Services transport layer.

The option information buffer conveys protocol information and quality-of-service parameters. The option information buffer structure, `t_optmgmt`, may be passed as an argument in these connection establishment and connectionless-mode data-transfer TLI functions:

- `t_connect`
- `t_listen`
- `t_accept`
- `t_rcvconnect`
- `t_sndudata`
- `t_rcvudata`

The `t_optmgmt` structure, defined in <tiuser.h>, is:

```
struct t_optmgmt {
    struct netbuf opt;
    long flags;
};
```

The structure required for the option information buffer `netbuf`, defined in <tiuser.h>, is:

```
struct netbuf {
    unsigned int maxlen;
    unsigned int len;
    char  *buf;
};
```

The `maxlen` member of this structure should contain the exact size of the buffer pointed to by `buf`.

The `len` member should contain the actual number of bytes used in the buffer.

The `buf` member contains a pointer to the option information structure, which contains connection establishment options. This buffer has no standard structure; the format of the options are implementation-dependent. The Internet Services Transport option format allows multiple options to be chained together in one request. The format for each option contains four fields:

- Option length word—Identifies the number of bytes this option structure contains
- Option level—Identifies the target level of the option. Levels supported are:
  - INET_GENERIC
  - INET_TCP
  - INET_UDP
  - INET_IP
- Option name—Indentifies the option to be managed
- Option value—Contains the value of the option and is at least one word long, more if needed to pass the option values

The C structure definition for the Internet Services option buffer format is:

```
struct t_option {
   unsigned long len;
   unsigned long level;
   unsigned long name;
   unsigned long value [1];
};
```

The option information buffer required by the Internet Services Transport protocol is illustrated in Figure 61.

Figure 61 Option information buffer format

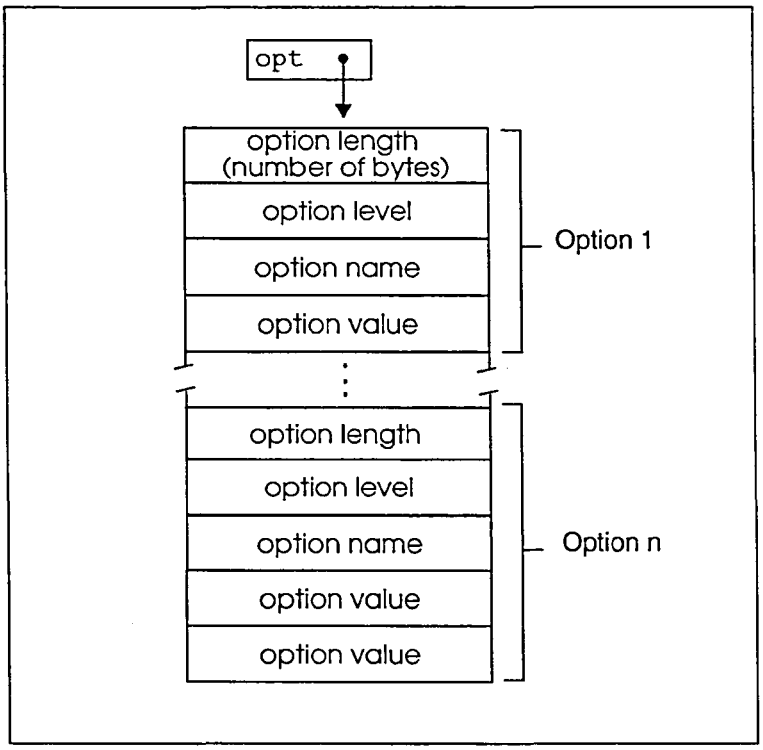
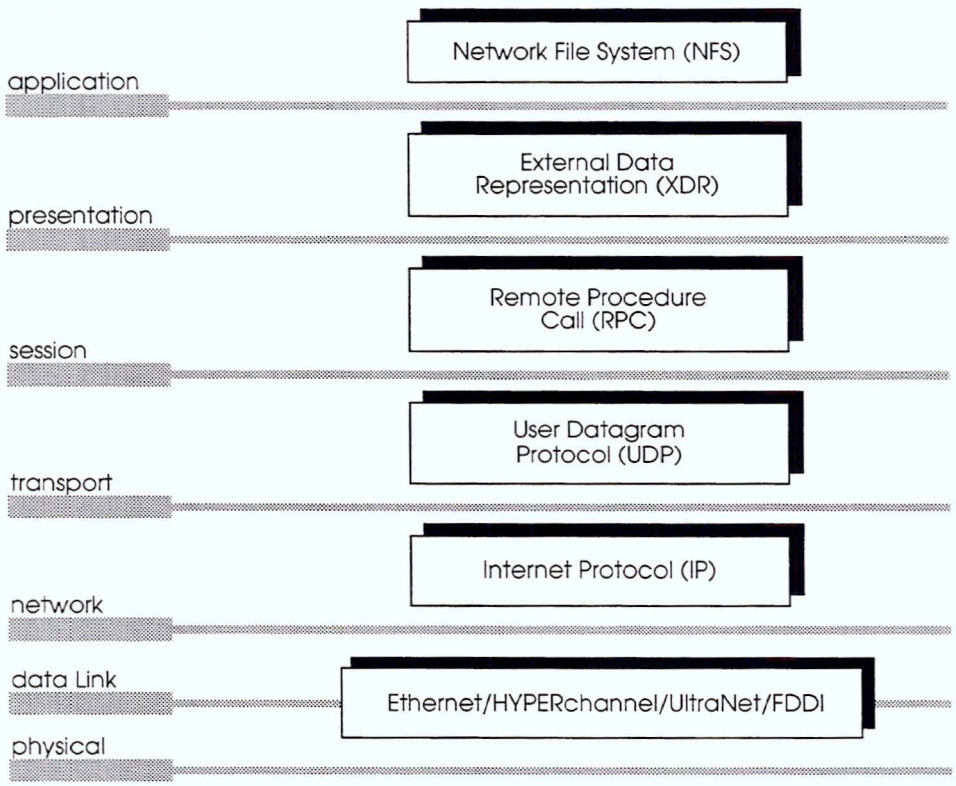


Table 19 lists options currently supported by Internet Services protocols.

Table 19 Options supported by Internet Services protocols

Option level	Option name	Used by TCP	Used by UDP
INET_GENERIC	SO_BROADCAST	x	x
	SO_DEBUG	x	x
	SO_DONTROUTE	x	x
	SO_LINGER	x	
	SO_KEEPALIVE	x	
	SO_RCVBUF	x	x
	SO_SNDBUF	x	x
	SO_SNDLOWAT	x	x
	SO_RCVLOWAT	x	x
	SO_REUSEADDR	x	x
	SO_USELOOPBACK	x	x
INET_TCP	TCP_NODELAY	x	
	TCP_MAXSEG	x	
INET_UDP	UDPCHECKSUM		x
INET_IP	IP_TOS	x	x
	IP_TTL	x	x
	IP_OPTIONS	x	x

NFS interface



NFS interface

CONVEX Network File System (NFS) provides transparent access to file systems on remote machines. NFS, developed by Sun Microsystems, links heterogeneous systems—including personal computers, workstations, supercomputers, and mainframes—to share resources and files over local area and wide area networks. File sharing is accomplished by mounting a remote file system, then reading or writing files in place.

CONVEX NFS includes two application program interfaces: Remote Procedure Call (RPC) and eXternal Data Representation (XDR). The NFS Protocol Specification is implemented using RPC and XDR. These facilities are also available for use in implementing distributed systems. CONVEX NFS also includes `rpcgen`, a compiler for the RPC language.

This part includes the following information on RPC and XDR programming:

- Chapter 14 provides an overview of NFS, RPC, and XDR
- Chapter 15 explains how to use `rpcgen` to develop RPC applications
- Chapter 16 describes RPC programming techniques
- Chapter 17 describes XDR programming techniques
- Chapter 18 is the NFS protocol specification (RFC 1094)
- Chapter 19 is the RPC protocol specification (RFC 1057)
- Chapter 20 is the XDR protocol specification (RFC 1014)
- Chapter 21 is the NIS protocol specification

Introduction to NFS, RPC, and XDR

14

Network File System (NFS)

NFS provides access to remote file systems in a heterogeneous network. Through use of Remote Procedure Call (RPC) primitives built on top of an External Data Representation (XDR), the NFS protocol is machine, operating system, network architecture, and transport protocol independent. In the CONVEX implementation, UDP provides the transport layer.

NFS software consists of a modified kernel, a set of library routines, and utility commands. NFS is not a distributed operating system, but rather, an interface that allows a variety of machines and operating systems to play the role of client or server.

NFS services are transparent to users; there is no syntactical difference between reading or writing a file on a local disk and reading or writing a file on a disk attached to a remote machine. The NFS server exports shared files so that client machines can access them. Client machines request access by mounting the exported file systems.

The NFS protocol is stateless. A server does not need to maintain any extra state information about any of its clients in order to function correctly. Stateless servers have a distinct advantage over state-oriented servers in the event of a failure. With stateless servers, a client need only retry a request until the server responds; it does not even need to know that the server has crashed or that the network temporarily went down. The client of a state-oriented server, on the other hand, needs to either detect a server crash and rebuild the server's state when it comes back up, or cause client operations to fail.

For more information about NFS services, configuration, and management, refer to *CONVEX Networking Concepts* and *Managing CONVEX Internet Services and NFS*.

Protocols and standards conformance

The CONVEX implementation of NFS is based on SUN NFSSRC 4.0, which conforms to the following RFCs for NFS, RPC, and XDR:

- RFC 1094, NFS: Network File System Protocol Specification
- RFC 1057, RPC: Remote Procedure Call Protocol Specification Version 2
- RFC 1014, XDR: External Data Representation Standard

Software architecture

Traditional file systems are composed of directories and files, each of which has a corresponding index node (inode) containing information about the file, such as location, permissions, and access times. inodes are assigned unique numbers within a file system, but a file on one file system could have the same inode number as a file on another file system. This presents a problem in a network environment.

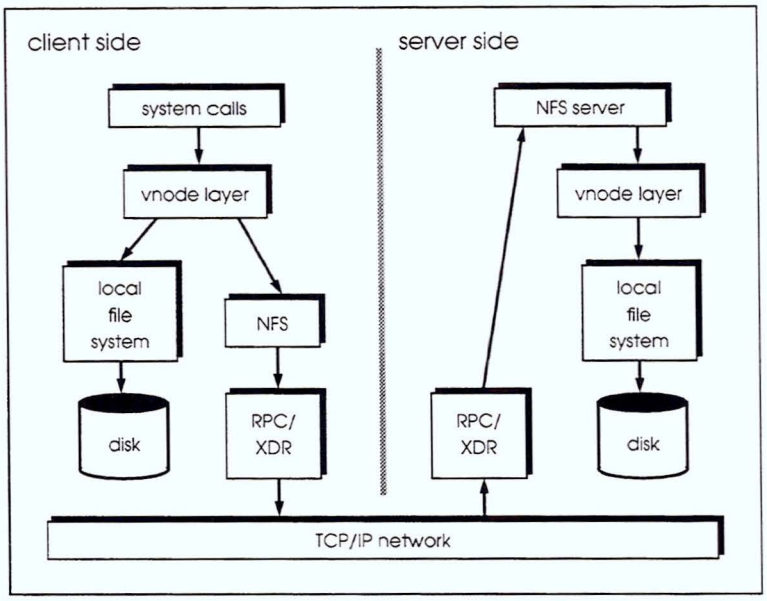
To solve this problem, CONVEX NFS is implemented through the virtual file system (VFS). VFS is based on the vnode, an additional layer in the file system structure that permits multiple types of file systems. VFS architecture divides the file subsystem into architecture-dependent and -independent layers, and provides a generic interface to the file system. This generic interface enables ConvexOS to accommodate local file systems as well as networked file systems.

Because ConvexOS file systems are manipulated through the VFS interface, the underlying file system implementation is transparent. Above the VFS interface, ConvexOS deals in vnodes; below this interface, the file system may or may not implement inodes.

A local VFS connects to file system data on a local device. The remote VFS defines and implements the NFS interface, using the remote procedure call (RPC) mechanism. RPC permits communication with remote services in a manner similar to the procedure calling mechanism available in many programming languages. RPC protocols are described using the eXternal Data Representation (XDR) package. XDR permits machine-independent representation and definition of high-level protocols on the network.

Figure 62 on page 207 illustrates NFS network architecture.

Figure 62 NFS network architecture



The arrows in Figure 62 indicate the flow of a request for data from a file system.

On the client side:

- For access through a local virtual file system, the request is directed to file system data on devices connected to the client machine.
- For access through a remote virtual file system, the request passes through the RPC/XDR layers to the network.

On the server side, the request passes through the RPC/XDR layers to the NFS server. vnodes are used to access a local file system on the NFS server and service the request.

The path of the data request is retraced to return results.

Remote Procedure Call (RPC)

The remote procedure call specification provides a procedure-oriented interface to remote services. Each server supplies a "program" that is a set of procedures. NFS is one such program. The combination of host address, program number, version number, and procedure number specifies one remote service procedure.

The RPC facility enables a client process to have another process execute a procedure call, as if the caller had executed the procedure call in its own address space. Because caller and server are separate processes, they can reside on different machines.

The RPC mechanism is implemented as a library of procedures and a specification for portable data transmission, known as eXternal Data Representation (XDR). RPC does not depend on services provided by specific protocols, so it can be used with any underlying transport protocol.

RPC is discussed in detail in "Remote Procedure Calls: Protocol Specification" on page 345 and "RPC programming" on page 237.

`rpcgen`

The details of programming applications that use RPC can be overwhelming. Perhaps most daunting is writing XDR routines to convert procedure arguments and results into their network format and vice versa.

The `rpcgen` compiler helps programmers write RPC applications simply and directly. `rpcgen` handles most of the lower-level details, allowing programmers to debug the main features of their applications, instead of spending most of their time debugging network interface code.

`rpcgen` accepts a remote program interface definition written in RPC Language, and produces C language output. Developers compile and link `rpcgen` output files in the usual way. The developer writes server procedures in any language that observes ConvexOS C calling conventions and links them with the server skeleton produced by `rpcgen` to get an executable server program.

To use a remote program, a programmer writes an ordinary main program that makes local procedure calls to the client stubs produced by `rpcgen`. Linking this program with `rpcgen` routines creates an executable program.

rpcgen reduces development time needed for coding and debugging low-level routines. All compilers, including rpcgen, do this at a small cost in efficiency and flexibility. However, many compilers allow escape hatches for programmers to mix low-level code with high-level code; rpcgen is no exception. In speed-critical applications, hand-written routines can easily be linked with rpcgen output. Also, programmers may use rpcgen output as a starting point and rewrite it as necessary.

The chapter "rpcgen programming" on page 211 explains how to use rpcgen to develop RPC applications.

External Data Representation (XDR)

The External Data Representation (XDR) standard consists of a set of library routines that provides a common way of representing data types over a network. XDR allows applications to transfer data between diverse machines, such as Sun Workstations, VAX, IBM-PC, and CONVEX machines.

XDR uses a language to describe intricate data formats in a concise manner. The language can be used only to describe data; it is not a programming language. XDR describes the most commonly used data types of high-level languages such as Pascal or C, so that applications written in these languages will be able to communicate easily over some medium.

The XDR standard makes the following assumption: that bytes (or octets), where a byte is defined to be 8 bits of data, are portable. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning.

XDR's approach to standardizing data representations is canonical; it defines a single byte order (Big Endian), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations. Similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents. Using a single standard completely decouples programs that create or send portable data from those that use or receive portable data.

For more information on XDR, refer to "XDR Standard: Technical Notes" on page 289 and "XDR Standard: Protocol Specification" on page 371.

rpcgen programming

15

The rpcgen protocol compiler

rpcgen helps programmers write RPC applications simply and directly. rpcgen does most of the dirty work, enabling programmers to debug the main features of their applications, instead of spending most of their time debugging network interface code.

rpcgen is a compiler that accepts a remote program interface definition written in a language called RPC Language, which is similar to C. It produces a C language output that includes stub versions of the following:

- Client routines
- Server skeleton
- XDR filter routines for both parameters and results
- Header file that contains common definitions

The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures invoked by remote clients. rpcgen output files can be compiled and linked in the usual way. The developer writes server procedures—in any language that observes ConvexOS "C" calling conventions—and links them with the server skeleton produced by rpcgen to get an executable server program. To use a remote program, a programmer writes an ordinary main program that makes local procedure calls to the client stubs produced by rpcgen. Linking this program with rpcgen stubs creates an executable program. (At present the main program must be written in C). rpcgen options can be used to suppress stub generation and to specify the transport to be used by the server stub.

Like all compilers, `rpcgen` reduces development time otherwise spent coding and debugging low-level routines. All compilers do this at a small cost in efficiency and flexibility. However, many compilers including `rpcgen` allow escape hatches for programmers to mix low-level code with high-level code. In speed-critical applications, hand-written routines can be linked with the `rpcgen` output with no difficulty. Also, you can use `rpcgen` output as a starting point, rewriting as necessary. (For a discussion of RPC programming without `rpcgen` refer to "RPC programming" on page 237.)

Converting local procedures into remote procedures

Assume an application that runs on a single machine, one which we want to convert to run over the network. Here we will demonstrate such a conversion by way of a simple example—a program that prints a message to the console shown in Figure 63.

Figure 63 Message printing example

```
/*
 * printmsg.c: print a message on the console
 */
#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    char *message;

    if (argc < 2) {
        fprintf(stderr, "usage: %s <message>\n", argv[0]);
        exit(1);
    }

    message = argv[1];

    if (!printmessage(message)) {
    fprintf(stderr, "%s: couldn't print your message\n", argv[0]);
    exit(1);
    }

    printf("Message Delivered!\n");
    exit(0);
}

/*
 * Print a message to the console.
 * Return a boolean indicating whether the message was
 * actually printed.
 */
printmessage(msg)
    char *msg;
{
    FILE *f;

    f = fopen("/dev/console", "w");
    if (f == NULL) {
        return (0);
    }
    fprintf(f, "%s\n", msg);
    fclose(f);
    return(1);
}
```

The program `printmsg.c` is compiled and run:

```
example% cc printmsg.c -o printmsg
example% printmsg "Hello, there."
Message delivered!
example%
```

If `printmessage()` was turned into a remote procedure, then it could be called from anywhere in the network. Ideally, one would just like to stick a keyword like *remote* in front of a procedure to turn it into a remote procedure. Unfortunately, we have to live within the constraints of the C language, since it existed before RPC did. Even without language support, it's not very difficult to make a procedure remote.

In general, it is necessary to determine the types for all procedure inputs and outputs. In this case, we have a procedure `printmessage()` which takes a string as input, and returns an integer as output. Knowing this, we can write a protocol specification in RPC language that describes the remote version of `printmessage`, shown below.

```
/*
 * msg.x: Remote message printing protocol
 */
program MESSAGEPROG {
   version MESSAGEVERS {
      int PRINTMESSAGE(string) = 1;
   } = 1;
} = 99;
```

Remote procedures are part of remote programs, so we actually declared an entire remote program here which contains the single procedure `PRINTMESSAGE`. This procedure is declared to be in version 1 of the remote program. Null procedure (procedure 0) is unnecessary because `rpcgen` generates it automatically.

Notice that procedures and variables are declared with all capital letters. This is not required, but is a good convention to follow.

Notice also that the argument type is string and not char *. This is because a char * in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a string.

There are just two more things to write. First, there is the remote procedure itself. Figure 64 shows the definition of a remote procedure to implement the `PRINTMESSAGE` procedure we declared above:

Figure 64 Remote version of `printmessage` procedure

```
    /*
     * msg_proc.c: implementation of the remote
     * procedure "printmessage"
     */
#include <stdio.h>
#include <rpc/rpc.h>     /* always needed */
#include "msg.h"         /* need this too: msg.h will be */
                         /* generated by rpcgen */

/*
 * Remote verson of "printmessage"
 */
int *
printmessage_1(msg)
    char **msg;
{
    static int result;  /* must be static! */
    FILE *f;

    f = fopen("/dev/console", "w");

    if (f == NULL) {
        result = 0;
       return (&result);
    }

    fprintf(f, "%s\n", *msg);
    fclose(f);
    result = 1;
    return (&result);
}
```

Notice that the declaration of the remote procedure `printmessage_1()` differs from that of the local procedure `printmessage()` in three ways:

- It takes a pointer to a string instead of a string itself. This is true of all remote procedures; they always take pointers to their arguments rather than the arguments themselves.
- It returns a pointer to an integer instead of an integer itself. This is also generally true of remote procedures. They always return a pointer to their results.
- It has an "_1" appended to its name. In general, all remote procedures called by `rpcgen` are named by the following rule: the name in the program definition (in this case

PRINTMESSAGE) is converted to all lower-case letters. An underbar ("_") is appended to it, and finally the version number (in this case 1) is appended.

Finally, we declare the main client program that will call the remote procedure as shown in Figure 65.

Figure 65 Main client program for message printing example

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include <rpc/rpc.h>      /* always needed */
#include "msg.h"          /* need this too: msg.h will be */
                          /* generated by rpcgen          */

main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    int *result;
    char *server;
    char *message;

    if (argc < 3) {
        fprintf(stderr, "usage: %s host message\n", argv[0]);
        exit(1);
    }

    /*
     * Save values of command line arguments
     */
    server = argv[1];
    message = argv[2];

    /*
     * Create client "handle" used for calling MESSAGEPROG
     * on the server designated on the command line.
     * We tell the RPC package to use the "tcp" protocol
     * when contacting the server.
     */
    cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
```

Figure 65 Main client program for message printing example **(continued)**

```
    if (cl == NULL) {
        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }
    /*
     * Call the remote procedure "printmessage" on the
     * server
     */
    result = printmessage_1(&message, cl);

    if (result == NULL) {

    /*
     * An error occurred while calling the server.
     * Print error message and die.
     */
        clnt_perror(cl, server);
        exit(1);
    }

    /*
     * Okay, we successfully called the remote procedure.
     */
    if (*result == 0) {

    /*
     * Server was unable to print our message.
     * Print error message and die.
     */
        fprintf(stderr, "%s: %s couldn't print your message\n", argv[0],
                server);
        exit(1);
    }

    /*
     * The message got printed on the server's console
     */
    printf("Message delivered to %s!\n", server);
}
```

There are two things to note here:

- First, a client "handle" is created using the RPC library routine `clnt_create`. This client handle is passed to the stub routines which call the remote procedure.
- The remote procedure `printmessage_1()` is called the same way as it is declared in `msg_proc.c`, except for the inserted client handle as the second argument.

Here is how to put all of the pieces together:

```
example% rpcgen msg.x
example% cc rprintmsg.c msg_clnt.c -o rprintmsg
example% cc msg_proc.c msg_svc.c -o msg_server
```

Two programs are compiled: the client program `rprintmsg` and the server program `msg_server`. Before doing this, `rpcgen` is used to fill in the missing pieces.

`rpcgen` created the following files and routines with the input file msg.x:

- Header file msg.h that contains #define for MESSAGEPROG, MESSAGEVERS, and PRINTMESSAGE for use in other modules.
- Client "stub" routines in the msg_clnt.c file. In this case there is only one, the `printmessage_1()` that was referred to from the `printmsg` client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is FOO.x, the client stubs output file is called FOO_clnt.c.
- Created the server program which calls `printmessage_1()` in msg_proc.c. This server program is named msg_svc.c. The rule for naming the server output file is similar to the previous one: for an input file called FOO.x, the output server file is named FOO_svc.c.

Copy the server to a remote machine and run it. For this example, the machine is called "moon." Server processes are run in the background because they never exit.

```
moon% msg_server &
```

Then, on our local machine, we can print a message on moon's console.

```
sun% printmsg moon "Hello, moon."
```

The message will be printed to moon's console. You can print a message on anybody's console (including your own) with this program if you are able to copy the server to their machine and run it.

Generating XDR routines

The previous example only demonstrated the automatic generation of client and server RPC code. `rpcgen` may also be used to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice-versa. This example presents a complete RPC service—a remote directory listing service, which uses `rpcgen` not only to generate stub routines, but also to generate the XDR routines. Figure 66 shows the protocol description file.

Figure 66 Protocol description file for remote directory listing service

```
/*
 * dir.x: Remote directory listing protocol
 */

/* maximum length of a directory entry */
const MAXNAMELEN = 255;

typedef string nametype<MAXNAMELEN>; /* a directory entry */
typedef struct namenode *namelist;   /* a link in the listing */

/*
 * A node in the directory listing
 */
struct namenode {
    nametype name;                /* name of directory entry */
    namelist next;                /* next entry */
};

/*
 * The result of a READDIR operation.
 */
union readdir_res switch (int errno) {
    case 0:
        namelist list;          /* no error: return directory listing */
    default:
        void;                   /* error occurred: nothing else to return */
};

/*
 * The directory program definition
 */
program DIRPROG {
    version DIRVERS {
        readdir_res READDIR(nametype) = 1;
    } = 1;
} = 76;
```

Note

Types (like readdir_res in the example above) can be defined using the "struct," "union," and "enum" keywords, but these keywords should not be used in subsequent declarations of variables of those types. For example, if you define a union "foo," you should declare using only "foo" and not "union foo." In fact, `rpcgen` compiles RPC unions into C structures, and it is an error to declare them using the "union" keyword.

Running `rpcgen` on dir.x creates four output files. Three are the same as before: header file, client stub routines and server skeleton. The fourth is the XDR routines necessary for converting the data types we declared into XDR format and vice-versa. These are output in the file dir_xdr.c.

Figure 67 shows the implementation of the READDIR procedure.

Figure 67 Remote readdir implementation

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>
#include <sys/dir.h>
#include "dir.h"

extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(dirname)
    nametype *dirname;
{
    DIR *dirp;
    struct direct *d;
    namelist nl;
    namelist *nlp;
    static readdir_res res; /* must be static! */

    /*
     * Open directory
     */
    dirp = opendir(*dirname);

    if (dirp == NULL) {
        res.errno = errno;
        return (&res);
    }
```

Figure 67 Remote readdir implementation (continued)

```
    /*
     * Free previous result
     */
    xdr_free(xdr_readdir_res, &res);

    /*
     * Collect directory entries.
     * Memory allocated here will be freed by xdr_free
     * next time readdir_1 is called
     */
    nlp = &res.readdir_res_u.list;

    while (d = readdir(dirp)) {
        nl = *nlp = (namenode *) malloc(sizeof(namenode));
        nl->name = strdup(d->d_name);
        nlp = &nl->next;
    }

    *nlp = NULL;

    /*
     * Return the result
     */
    res.errno = 0;
    closedir(dirp);
    return (&res);
}
```

Finally, the client side program to call the server is shown in Figure 68.

Figure 68 Remote directory listing client

```
/*
 * rls.c: Remote directory listing client
 */
#include <stdio.h>
#include <rpc/rpc.h> /* always need this */
#include "dir.h"     /* will be generated by rpcgen */

extern int errno;
main(argc, argv)
    int argc;
    char *argv[];
{
    CLIENT *cl;
    char *server;
    char *dir;
    readdir_res *result;
    namelist nl;

    if (argc != 3) {
        fprintf(stderr, "usage: %s host directory\n",
            argv[0]);
    exit(1);
    }
    /*
     * Remember what our command line arguments refer to
     */
    server = argv[1];
    dir = argv[2];

    /*
     * Create client "handle" used for calling DIRPROG
     * on the server designated on the command line. We tell
     * the RPC package to use the "tcp" protocol when
     * contacting the server.
     */
    cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");

    if (cl == NULL) {

        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
    }
```

Figure 68 Remote directory listing client **(continued)**

```
    /*
     * Call the remote procedure readdir on the server
     */
    result = readdir_1(&dir, cl);

    if (result == NULL) {

        /*
         * An error occurred while calling the server.
         * Print error message and die.
         */
        clnt_perror(cl, server);
        exit(1);
    }

    /*
     * Okay, we successfully called the remote procedure.
     */
    if (result->errno != 0) {

        /*
         * A remote system error occurred.
         * Print error message and die.
         */
        errno = result->errno;
        perror(dir);
        exit(1);
    }

    /*
     * Successfully got a directory listing.
     * Print it out.
     */
    for (nl = result->readdir_res_u.list; nl != NULL;
        nl = nl->next) {
        printf("%s\n", nl->name);
    }

    exit(0);
}
```

The following lines show the commands to compile and run the programs, and the displayed output:

```
sun%  rpcgen dir.x
sun% cc rls.c dir_clnt.c dir_xdr.c -o rls
sun% cc dir_svc.c dir_proc.c dir_xdr.c -o
dir_svc
sun% dir_svc &
moon% rls sun /usr/pub
.
..
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
moon%
```

A final note about `rpcgen`: the client program and the server procedure can be tested together as a single program by simply linking them with each other rather than with the client and server stubs. The procedure calls will be executed as ordinary local procedure calls and the program can be debugged with a local debugger such as `dbx`. When the program is working, the client program can be linked to the client stub produced by `rpcgen`, and the server procedures can be linked to the server stub produced by `rpcgen`.

Note

If you do this, you may want to comment out calls to RPC library routines and have client-side routines call server routines directly.

The C preprocessor

The C preprocessor is run on all input files before they are compiled, so all the preprocessor directives are legal within a ".x" file. Four symbols can be defined, depending upon which output file is getting generated. The symbols are shown in Table 20.

Table 20 C preprocessor symbols

Symbol	Use
RPC_HDR	Header-file output
RPC_XDR	XDR routine output
RPC_SVC	Server-skeleton output
RPC_CLNT	Client stub output

In addition, rpcgen does some preprocessing. Any line that begins with a percent sign is passed directly into the output file without any interpretation. Figure 69 demonstrates the preprocessing features.

Figure 69 Preprocessor example

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
    } = 1;
} = 44;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
%       static int thetime;
%
%       thetime = time(0);
%       return (&thetime);
%}
#endif
```

The "%" feature is not generally recommended, as there is no guarantee that the compiler will place the output where you intended.

rpcgen programming notes

This section explains how rpcgen handles timeouts, broadcast RPC, and additional information passed to server procedures.

Timeout changes

RPC sets a default timeout of 25 seconds for RPC calls when clnt_create() is used. This timeout may be changed using clnt_control(). Here is a small code fragment to demonstrate use of clnt_control().

```
struct timeval tv;

CLIENT *cl;
cl = clnt_create("somehost", SOMEPROG,
   SOMEVERS, "tcp");

if (cl == NULL) {
   exit(1);
}

tv.tv_sec = 60; /*
                 *  change timeout to 1
                 *  minute
                 */
tv.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

Handling broadcast on the server side

When a procedure is known to be called via broadcast RPC, it is usually wise for the server to not reply unless it can provide some useful information to the client. This prevents the network from getting flooded by useless replies.

To prevent the server from replying, a remote procedure can return NULL as its result, and the server code generated by rpcgen will detect this and not send out a reply.

Here is an example of a procedure that replies only if the remote procedure determines that the local system is running the NFS server program:

```
void *
reply_if_nfsserver()
{
    char notnull;  /*
                    * just here so we can use
                    * its address
                    */

    if (access("/etc/exports", F_OK) < 0) {
        return (NULL);     /*
                            * prevent RPC from
                            * replying
                            */
    }
    /*
     * return non-null pointer so RPC will
     * send out a reply
     */
    return ((void *)&notnull);
}
```

Note

If procedures return type "void *," they must return a non-NULL pointer if they want RPC to reply for them.

Other information passed to server procedures

Server procedures will often want to know more about an RPC call than just its arguments. For example, getting authentication information is important to procedures that want to implement some level of security. This extra information is actually supplied to the server procedure as a second argument. Here is an example to demonstrate its use. What we've done here is rewrite the previous `printmessage_1()` procedure to only allow root users to print a message to the console.

```
int *
printmessage_1(msg, rq)
   char **msg;
   struct svc_req *rq;
{
   static in result; /* Must be static */
   FILE *f;
   struct suthunix_parms *aup;

   aup = (struct authunix_parms *)
      rq->rq_clntcred;

   if (aup->aup_uid != 0) {
      result = 0;
      return (&result);
   }

   /*
    * Same code as before.
    */
}
```

RPC language

RPC language is an extension of XDR language. The sole extension is the addition of the program type. For a complete description of the XDR language syntax, refer to "XDR Standard: Protocol Specification" on page 371. For a description of the RPC extensions to the XDR language, refer to "Remote Procedure Calls: Protocol Specification" on page 345.

The XDR language is so close to C that if you know C, you know most of it already. We describe here the syntax of the RPC language, showing a few examples along the way. We also show how the various RPC and XDR type definitions get compiled into C-type definitions in the output header file.

Definitions

An RPC language file consists of a series of definitions, as shown below.

```
definition-list:
   definition ";"
   definition ";" definition-list
```

It recognizes five types of definitions:

```
definition:
   enum-definition
   struct-definition
   union-definition
   typedef-definition
   const-definition
   program-definition
```

Structures

An XDR struct is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
   "struct" struct-ident "{"
      declaration-list
   "}"

declaration-list:
   declaration ";"
   declaration ";" declaration-list
```

The following example shows an XDR structure to a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file.

```
struct coord {          struct coord {
   int x;                  int x;
   int y;          ➜       int y;
};                      };
                        typedef struct coord
                           coord;
```

The output is identical to the input, except for the added typedef at the end of the output. This allows one to use "`coord`" instead of "`struct coord`" when declaring items.

Unions

XDR unions are discriminated unions, and look quite different from C unions. They are more analogous to Pascal variant records than they are to C unions.

```
union-definition:
   "union" union-ident "switch" "("
      declaration ")" "{"
      case-list
   "}"

case-list:
   "case" value ":" declaration ";"
   "default" ":" declaration ";"
   "case" value ":" declaration ";" case-list
```

Here is an example of a type that might be returned as the result of a "read data" operation. If there is no error, then it returns a block of data. Otherwise, it returns nothing.

```
   union read_result switch (int errno) {
case 0:
   opaque data[1024];
default:
   void;
};
```

It gets compiled into the following:

```
struct read_result {
   int errno;
   union {
      char data[1024];
   } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the name as the type name, except for the trailing "_u."

Enumerations

XDR enumerations have the same syntax as C enumerations, as shown.

```
enum-definition:
   "enum" enum-ident "{"
      enum-value-list
   "}"

enum-value-list:
   enum-value
   enum-value "," enum-value-list
```

```
enum-value:
   enum-value-ident
   enum-value-ident "=" value
```

Here is a short example of an XDR enum, and the C enum that it gets compiled into.

```
enum colortype {          enum colortype {
   RED = 0,                  RED = 0,
   GREEN = 1,        ➜       GREEN = 1,
   BLUE = 2                  BLUE = 2,
};                        };
                          typedef enum
                             colortype colortype;
```

Typedef

XDR typedefs have the same syntax as C typedefs.

```
typedef-definition:
   "typedef" declaration
```

Here is an example that defines a `fname_type` used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; typedef char
   *fname_type;
```

Constants

XDR constants are symbolic constants that may be used wherever an integer constant is used, for example, in array size specifications.

```
const-definition:
   "const" const-ident "=" integer
```

For example, the following defines a constant `DOZEN` equal to 12:

```
const DOZEN = 12;      ➜      #define DOZEN 12
```

Programs

RPC programs are declared using the following syntax:

```
program-definition:
   "program" program-ident "{"
      version-list
   "}" "=" value

version-list:
   version ";"
   version ";" version-list

version:
   "version" version-ident "{"
      procedure-list
   "}" "=" value

procedure-list:
   procedure ";"
   procedure ";" procedure-list

procedure:
   type-ident procedure-ident "(" type-ident
   ")" "=" value
```

For example, here is the time protocol, revisited.

```
/*
 * time.x: Get or set the time. Time is
 * represented as number
 * of seconds since 0:00, January 1, 1970.
 */
program TIMEPROG {
   version TIMEVERS {
      unsigned int TIMEGET(void) = 1;
      void TIMESET(unsigned) = 2;
   } = 1;
} = 44;
```

This file compiles into `#define` lines in the output header file.

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

Declarations

In XDR, there are only four kinds of declarations.

- simple-declaration:

  Simple-declarations are just like simple C declarations.

```
type-ident variable-ident
```

Example:

```
colortype color;  ➜   colortype color;
```

- fixed-array-declaration:

  Fixed-length Array Declarations are just like C array declarations.

  ```
  type-ident variable-ident "[" value "]"
  ```

  Example:

  ```
  colortype palette[8];➜ colortype palette[8];
  ```

- variable-array-declaration:

  Variable-Length Array Declarations have no explicit syntax in C, so XDR invents its own using angle-brackets.

  ```
  type-ident variable-ident "<" value ">"
  type-ident variable-ident "<" ">"
  ```

  The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size.

  ```
  int heights<12>; /* at most 12 items */
  int widths<>; /* any number of items */
  ```

  Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into structures. For example, the "heights" declaration gets compiled into the following structure:

  ```
  struct {
     u_int heights_len; /* # of items in
                         * array */
     int *heights_val;   /* pointer to
                          * array
                          */
  } heights;
  ```

Note

The number of items in the array is stored in the "_len" component and the pointer to the array is stored in the "_val" component. The first part of each of these component's names is the same as the name of the declared XDR variable.

- pointer-declaration:

  Pointer Declarations are made in XDR exactly as they are in C. You can't really send pointers over the network, but you can use XDR pointers for sending recursive data types such

as lists and trees. The type is actually called "optional-data", not "pointer", in XDR language.

```
type-ident "*" variable-ident
```

Example:

```
listitem *next;    ➜ listitem *next;
```

Special cases

There are a few exceptions to the rules previously described. These exceptions are examined below.

Booleans

C has no built-in boolean type. However, the RPC library has a boolean type called `bool_t` that is either `TRUE` or `FALSE`. Variables declared as type `bool` in XDR language are compiled into `bool_t` in the output header file.

Example:

```
bool married;    ➜ bool_t married;
```

Strings

C has no built-in string type, but instead uses the null-terminated `char *` convention. In XDR language, strings are declared using the `string` keyword, and compiled into `char *` in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the NULL character). The maximum size can be left off, indicating a string of arbitrary length.

Examples:

```
string name<32>;          char *name;
string longname<>;    ➜   char *longname;
```

Opaque data

Opaque data is used in RPC and XDR to describe untyped data, that is, sequences of arbitrary bytes. It can be declared either as a fixed or variable length array.

Examples:

```
opaque diskblock[512];      char diskblock[512];

opaque filedata<1024>;➜     struct {
                               u_int filedata_len;
                               char *filedata_val;
                            } filedata;
```

Voids

In a void declaration, the variable is not named. The declaration is simply "void" and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure).

RPC programming

16

This chapter explains how to write network applications using remote procedure calls, and describes the RPC mechanisms usually hidden by the `rpcgen` protocol compiler. `rpcgen` is described in detail in "rpcgen programming" on page 211.

Before attempting to write a network application or convert an existing nonnetwork application to run over the network, you should understand the material in this chapter. However, for most applications, you can circumvent the need to cope with the details presented here by using `rpcgen`. "Generating XDR routines" on page 219 contains the complete source for a working RPC service—a remote directory listing service which uses `rpcgen` to generate XDR routines as well as client and server stubs.

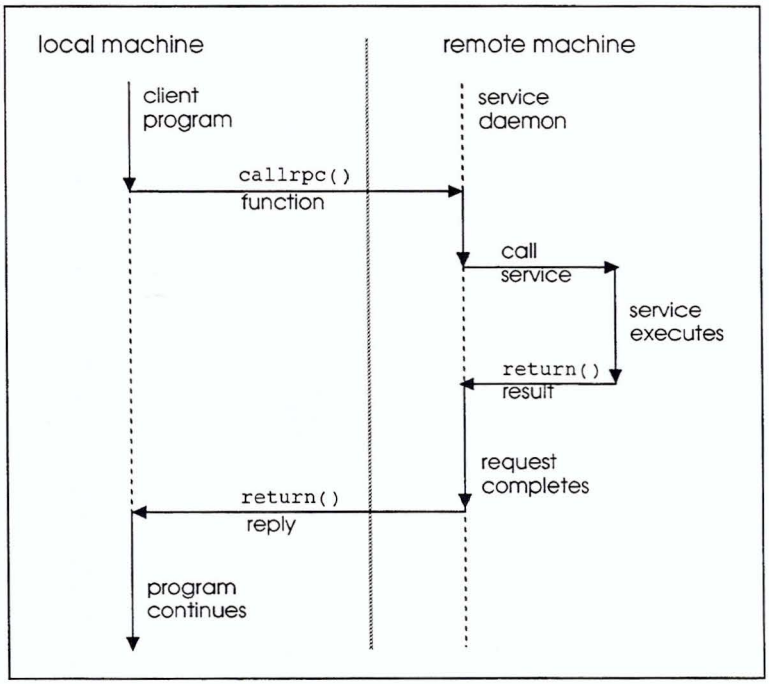
Although this chapter only discusses the interface to C, remote procedure calls can be made from any language. This chapter focuses on RPC in a networked environment; however, RPC can also be used for communication between different processes on the same machine.

The RPC protocol is specified in RFC 1057, Remote Procedure Call Protocol Specification Version 2. Instructions for obtaining RFCs are provided in *CONVEX Networking Concepts*, order number DSW-128.

Introduction to RPC

RPC enables communication with remote services in a manner similar to the local procedure calling mechanism available in many programming languages. Figure 70 shows a model of the remote procedure call mechanism.

Figure 70 RPC model



In the local procedure call model, the caller places arguments to a procedure in a specified location, then transfers control to the procedure. When the procedure completes, the caller regains control. At that point, the caller extracts the results from the specified location and continues execution.

The remote procedure call is similar, in that one thread of control logically winds through two processes—one is the caller's process, the other is the server's process. The caller sends a call message that includes the procedure's parameters to the server and waits for a reply. The reply includes the procedure's results. After the caller receives a reply, it extracts the results and resumes execution.

On the server side, a dormant server awaits the arrival of a call message. When a call message arrives, the server extracts the procedure's parameters, computes the results, sends a reply, and awaits the next call message.

Each RPC server supplies a program made up of a set of procedures described using RPC Language, an extension to the XDR language. The combination of host address, program number, and procedure number uniquely identifies one remote service procedure.

Layers of RPC

The RPC interface can be thought of as divided into three layers.

Highest layer

This layer consists of RPC library-based services and is transparent to the operating system, machine, and network upon which it is run. This layer can be considered as a way of using RPC, rather than as a part of RPC proper. Programmers who write RPC routines should almost always make this layer available to others by way of a simple C front end that entirely hides the networking.

For example, at the highest level, users can simply make calls to `rnusers`, a C routine that returns the number of users on a remote machine. Users are not explicitly aware of using RPC—they simply call a procedure, just as they would call `malloc`, for instance.

Middle layer

The middle layer is really "RPC proper." Here, users don't need to consider details about sockets, the operating system, or other low-level implementation mechanisms. They simply make remote procedure calls to routines on other machines. The middle-layer routines are used for most applications.

RPC calls are made with the system routines `registerrpc`, `callrpc`, and `svc_run`. `registerrpc` obtains a unique system-wide procedure-identification number, and `callrpc` actually executes a remote procedure call. At the middle level, a call to `rnusers` is implemented by these two routines.

The middle layer is rarely used in serious programming due to its inflexibility (simplicity). It does not allow timeout specifications or the choice of transport. It allows no UNIX process control or flexibility in case of errors. It doesn't support multiple kinds of call authentication. The programmer rarely needs all these kinds of control, but one or two of them is often necessary.

Lowest layer

The lowest layer does enable you to control details such as choice of transport or type of authentication. Programs written at this level are most efficient and flexible. Lowest layer routines include client creation routines such as `clnt_create`, the actual client call `clnt_call`, server creation routines such as `svcudp_create`, and the server registration routine `svc_register`.

Example of RPC higher layer

Imagine you're writing a program that needs to know how many users are logged into a remote machine. You can do this by calling the RPC library routine `rnusers` as illustrated in Figure 71:

Figure 71 RPC library routine `rnusers`

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char **argv;
{
    int num;

    if (argc != 2) {
        fprintf(stderr, "usage: rnusers hostname\n");
        exit(1);
    }

    if ((num = rnusers(argv[1])) < 0) {
        fprintf(stderr, "error: rnusers\n");
        exit(-1);
    }
    printf("%d users on %s\n", num, argv[1]);
    exit(0);
}
```

RPC library routines, such as `rnusers`, are in the RPC services library librpcsvc.a. Compile the program listed above by entering

```
% cc program.c -lrpcsvc
```

RPC library routines are documented in section 3 ConvexOS man pages. Refer to the intro(3R) man page for a complete list of the RPC service library functions and protocols. Table 21 lists some of the RPC service library routines available to the C programmer.

Table 21 RPC service library routines

Routine	Description
`rnusers()`	Return number of users on remote machine
`rusers()`	Return information about users on remote machine
`havedisk()`	Determine if remote machine has disk
`rstats()`	Get performance data from remote kernel
`rwall()`	Write to specified remote machines
`yppasswd()`	Update user password in Network Information Service (NIS)

Other RPC services — for example `ether()`, `mount()`, `rquota()`, and `spray()` are not available to the C programmer as library routines. They do, however, have RPC program numbers so they can be invoked with `callrpc()`, which is discussed in the next section. Most of them also have compilable rpcgen(1) protocol description files. (The `rpcgen` protocol compiler radically simplifies the process of developing network applications. See the chapter "rpcgen programming" on page 211 for detailed information about `rpcgen` and `rpcgen` protocol description files).

Intermediate layer

The simplest interface, which explicitly makes RPC calls, uses the functions `callrpc` and `registerrpc`. Using this method, the number of remote users can be determined as shown in Figure 72.

Figure 72 Determining number of remote users with `callrpc()` and `registerrpc()`

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

main(argc, argv)
    int argc;
    char **argv;
{
    unsigned long nusers;
    int stat;

    if (argc != 2) {
       fprintf(stderr, "usage: nusers hostname\n");
       exit(-1);
    }

    if (stat = callrpc(argv[1],
       RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
       xdr_void, 0, xdr_u_long, &nusers) != 0) {
          clnt_perrno(stat);
          exit(1);
}

    printf("%d users on %s\n", nusers, argv[1]);
    exit(0);
}
```

Each RPC procedure is uniquely defined by a program number, version number, and procedure number. The program number specifies a group of related remote procedures, each of which has a different procedure number. Each program also has a version number; so, when a minor change is made to a remote service (adding a new procedure, for example), a new program number doesn't have to be assigned. When you want to call a procedure to find the number of remote users, you look up the appropriate program, version, and procedure numbers in a manual, just as you look up the name of a memory allocator when you want to allocate memory.

The simplest way to make remote procedure calls is with the RPC library routine `callrpc`. The format of `callrpc` is

```
callrpc(host, prognum, versnum, procnum, inproc,
        in, outproc, out)
char *host;
u_long prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
```

where

`host`	is the name of the remote server machine
`prognum`	is the program number
`versnum`	is the version number
`procnum`	is the procedure number
`inproc`	Is used to encode the procedure's parameters
`in`	Is the address of the procedure's argument(s)
`outproc`	Is used for decoding the results returned by the remote procedure
`out`	Is the address where the results are placed

Multiple arguments and results are handled by embedding them in structures. `callrpc` returns zero if it succeeds, or the value of `enum clnt_stat` cast to an integer if it fails. The return codes are in the file <rpc/clnt.h>.

Since data types may be represented differently on different machines, `callrpc` needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). In Figure 72, the return value for `RUSERSPROC_NUM` is an unsigned long, so `callrpc` has `xdr_u_long` as its first return parameter. This indicates that the result is of type unsigned long, and `&nusers` is its second return parameter, which is a pointer to where the long result will be placed. Since `RUSERSPROC_NUM` takes no argument, the argument parameter of `callrpc` is `xdr_void`.

If `callrpc` gets no answer after trying several times to deliver a message, it returns an error code. The delivery mechanism is UDP. Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the RPC library, discussed later in this document. Figure 73 shows an example remote server procedure corresponding to the procedure in Figure 72.

Figure 73 Remote server procedure to determine number of remote users

```
char *
nuser(indata)

    char *indata;
{
    unsigned long nusers;
    /*
     * Code here to compute the number of users
     * and place result in variable nusers.
     */
    return((char *)&nusers);
}
```

The remote server procedure takes one argument, a pointer to the input of the remote procedure call (ignored in our example), and returns a pointer to the result. In the current version of C, character pointers are generic pointers, so both the input argument and the return value are cast to char *.

Normally, a server registers all of the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests. In this example, there is only a single procedure to register, so the main body of the server looks like the code shown in Figure 74:

Figure 74 Registering a procedure

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

char *nuser();

main()
{
    registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
       nuser, xdr_void, xdr_u_long);
    svc_run();
    /* Never returns */
    fprintf(stderr, "Error: svc_run returned!\n");
    exit(1);
}
```

The `registerrpc()` routine registers a C procedure as corresponding to a given RPC procedure number. The first three parameters, `RUSERPROG`, `RUSERSVERS`, and `RUSERSPROC_NUM` are the program, version, and procedure numbers, respectively, of the remote procedure to be registered. `nuser()` is the name of the local procedure that implements the remote procedure, and `xdr_void()` and `xdr_u_long()` are the XDR filters for the remote procedure's arguments and results, respectively. (Multiple arguments or multiple results are passed as structures.)

Only the UDP transport mechanism can use `registerrpc()`. Thus, it is always safe in conjunction with calls generated by `callrpc()`.

Note

The UDP transport mechanism can only deal with arguments and results less than 8K bytes in length.

After registering the local procedure, the server program's main procedure calls `svc_run()`, the RPC library's remote procedure dispatcher. It is this function that calls the remote procedures in response to RPC call messages. Note that the dispatcher takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered.

Assigning program numbers

Program numbers are assigned in groups of 0x20000000 according to Table 22:

Table 22 Program number assignments

Program Number	Description
0x0 - 0x1fffffff	Defined by Sun
0x20000000 - 0x3fffffff	Defined by user
0x40000000 - 0x5fffffff	Transient
0x60000000 - 0x7fffffff	Reserved
0x80000000 - 0x9fffffff	Reserved
0xa0000000 - 0xbfffffff	Reserved
0xc0000000 - 0xdfffffff	Reserved
0xe0000000 - 0xffffffff	Reserved

Sun Microsystems administers the first group of numbers, which should be identical for all Sun customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range. The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

To register a protocol specification, send a request by network mail to *rpc@sun* or write to:

RPC Administrator
Sun Microsystems
2550 Garcia Ave.
Mountain View, CA 94043

Please include a compilable `rpcgen` ".x" file describing your protocol. You will be given a unique program number in return.

The RPC program numbers and protocol specifications of standard Sun RPC services can be found in the include files in /usr/include/rpcsvc. These services, however, constitute only a small subset of those which have been registered. Table 23 is a complete list of registered programs, at the time of this printing.

Table 23 Registered RPC programs

RPC number	Program	Description
100000	PMAPPROG	portmapper
100001	RSTATPROG	remote stats
100002	RUSERSPROG	remote users
100003	NFSPROG	NFS
100004	YPPROG	NIS
100005	MOUNTPROG	mount daemon
100006	DBXPROG	remote dbx
100007	YPBINDPROG	YP binder
100008	WALLPROG	shutdown msg
100009	YPPASSWDPROG	yppasswd server
100010	ETHERSTATPROG	ether stats
100011	RQUOTAPROG	disk quotas
100012	SPRAYPROG	spray packets
100013	IBM3270PROG	3270 mapper
100014	IBMRJEPROG	RJE mapper
100015	SELNSVCPROG	selection service
100016	RDATABASEPROG	remote database access
100017	REXECPROG	remote execution
100018	ALICEPROG	Alice Office Automation
100019	SCHEDPROG	scheduling service

Table 23 Registered RPC programs **(continued)**

RPC number	Program	Description
100020	LOCKPROG	local lock manager
100021	NETLOCKPROG	network lock manager
100022	X25PROG	x.25 inr protocol
100023	STATMON1PROG	status monitor 1
100024	STATMON2PROG	status monitor 2
100025	SELNLIBPROG	selection library
100026	BOOTPARAMPROG	boot parameters service
100027	MAZEPROG	mazewars game
100028	YPUPDATEPROG	YP update
100029	KEYSERVEPROG	key server
100030	SECURECMDPROG	secure login
100031	NETFWDIPROG	NFS net forwarder init
100032	NETFWDTPROG	NFS net forwarder trans
100033	SUNLINKMAP_PROG	sunlink MAP
100034	NETMONPROG	network monitor
100035	DBASEPROG	lightweight database
100036	PWDAUTHPROG	password authorization
100037	TFSPROG	translucent file svc
100038	NSEPROG	nse server
100039	NSE_ACTIVATE_PROG	nse activate daemon

Table 23 Registered RPC programs **(continued)**

RPC number	Program	Description
150001	PCNFSDPROG	pc passwd authorization
200000	PYRAMIDLOCKINGPROG	Pyramid-locking
200001	PYRAMIDSYS5	Pyramid-sys5
200002	CADDS_IMAGE	CV cadds_image
300001	ADT_RFLOCKPROG	ADT file locking

Passing arbitrary data types

In the previous example, the RPC call passes a single unsigned long. RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a network standard called *External Data Representation* (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called serializing and the reverse process is called deserializing. The type field parameters of `callrpc()` and `registerrpc()` can be a built-in procedure like `xdr_u_long()` in Figure 74, or a user supplied one. XDR has these built-in type routines:

```
xdr_int()    xdr_u_int()     xdr_enum()
xdr_long()   xdr_u_long()    xdr_bool()
xdr_short()  xdr_u_short()   xdr_wrapstring()
xdr_char()   xdr_u_char()
```

Note

The routine `xdr_string()` exists, but cannot be used with `callrpc()` and `registerrpc()`, which only pass two parameters to their XDR routines. `xdr_wrapstring()` has only two parameters, and therefore can be used. It calls `xdr_string()`.

As an example of a user-defined type routine, if you want to send the structure

```
struct simple {
    int a;
    short b;
} simple;
```

you can call `callrpc()` through

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
   xdr_simple, &simple ...);
```

where `xdr_simple()` is

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)
   XDR *xdrsp;
   struct simple *simplep;
{
   if (!xdr_int(xdrsp, &simplep->a))
         return (0);

   if (!xdr_short(xdrsp, &simplep->b))
         return (0);

   return (1);
}
```

An XDR routine returns nonzero (true in the sense of C) if it completes successfully; otherwise, it returns 0.

The existing XDR routines handle both encoding and decoding in the same routine, and all XDR routines are expected to do the same. A complete description of XDR is in "XDR Standard: Protocol Specification" on page 371. Only a few implementation examples are given here.

In addition to the built-in primitives, there are also the following prefabricated building blocks:

```
xdr_array()     xdr_bytes()     xdr_reference()
xdr_vector()    xdr_union()     xdr_pointer()
xdr_string()    xdr_opaque()
```

To send a variable array of integers, you might package them up as a structure like this:

```
struct varintarr {
   int *data;
   int arrlnth;
} arr;
```

and make an RPC call such as

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
   xdr_varintarr, &arr...);
```

with *xdr_varintarr()* defined as:

```
xdr_varintarr(xdrsp, arrp)
   XDR *xdrsp;
```

```
struct varintarr *arrp; {
   return (xdr_array(xdrsp, &arrp->data,
      &arrp->arrlnth,
         MAXLEN, sizeof(int), xdr_int));
}
```

The `xdr_array` routine above takes the following as parameters:

- The XDR handle
- A pointer to the array
- A pointer to the size of the array
- The maximum allowable array size
- The size of each array element,
- An XDR routine for handling each array element

If the size of the array is known in advance, you can use `xdr_vector()`, which serializes fixed-length arrays.

```
int intarr[SIZE];
xdr_intarr(xdrsp, intarr)
   XDR *xdrsp;
   int intarr[];
{
int i;
return (xdr_vector(xdrsp, intarr, SIZE,
      sizeof(int),
     xdr_int));
}
```

XDR always converts quantities to 4-byte multiples when serializing. Thus, if either of the examples mentioned previously involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine `xdr_bytes()`, which is like `xdr_array()` except that it packs characters. `xdr_bytes()` has four parameters, similar to the first four parameters of `xdr_array()`. For null-terminated strings, there is also the `xdr_string()` routine, which is the same as `xdr_bytes()` without the length parameter. When serializing, it gets the string length from `strlen()`; and, when deserializing, it creates a null-terminated string.

Figure 75 shows an example of serializing that calls the previously written `xdr_simple()` routine, as well as the built-in functions `xdr_string()` and `xdr_reference()`, which chases pointers.

Figure 75 Serializing example

```
struct finalexample {
    char *string;
    struct simple *simplep;
} finalexample;

xdr_finalexample(xdrsp, finalp)
    XDR *xdrsp;
    struct finalexample *finalp;
{

    if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
       return (0);

    if (!xdr_reference(xdrsp, &finalp->simplep,
       sizeof(struct simple), xdr_simple);
       return (0);
    return (1);
}
```

Note `xdr_simple()` could be called instead of `xdr_reference()`.

Lowest layer of RPC

In the examples given so far, RPC takes care of many details automatically for you. This section shows how to change the defaults by using lower layers of the RPC library. It is assumed that you are familiar with sockets and the system calls for dealing with them.

There are several occasions when you may need to use lower layers of RPC:

- You may need to use TCP since the higher layer uses UDP, which restricts RPC calls to 8K bytes of data. Using TCP permits calls to send long streams of data. Refer to the section "TCP" on page 279 for an example.
- You may want to allocate and free memory while serializing or deserializing with XDR routines. There is no call at the higher level to let you free memory explicitly. For more explanation, refer to the section "Memory allocation with XDR" on page 257.
- You may need to perform authentication on either the client or server side, by supplying credentials or verifying them. Refer to the section "Authentication" on page 269 for details.

More on the server side

The server for the `nusers()` program shown in Figure 75 does the same thing as the one in Figure 74 that uses `registerrpc()`, but is written using a lower layer of the RPC package.

Figure 76 `nusers()` example using lower layer of RPC

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

main()
{
    SVCXPRT *transp;
    int nuser();

    transp = svcudp_create(RPC_ANYSOCK);

    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
          exit(1);
    }

    pmap_unset(RUSERSPROG, RUSERSVERS);

    if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser,
       IPPROTO_UDP)) {
       fprintf(stderr, "can't register RUSER service\n");
       exit(1);
    }

    svc_run();  /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}

nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;

    switch (rqstp->rq_proc) {
       case NULLPROC:
          if (!svc_sendreply(transp, xdr_void, 0))
             fprintf(stderr, "can't reply to RPC call\n");
          return;
       case RUSERSPROC_NUM:
             /* Code here to compute the number of users
              * and assign it to the variable nusers */
          if (!svc_sendreply(transp, xdr_u_long, &nusers))
             fprintf(stderr, "can't reply to RPC call\n");
          return;
    default:
          svcerr_noproc(transp);
          return;
    }
}
```

First, the server gets a transport handle, which is used for receiving and replying to RPC messages. `registerrpc()` uses `svcudp_create()` to get a UDP handle. If you require a more reliable protocol, call `svctcp_create()` instead. If the argument to `svcudp_create()` is `RPC_ANYSOCK`, the RPC library creates a socket on which to receive and reply to RPC calls. Otherwise, `svcudp_create()` expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of `svcudp_create()` and `clnttcp_create()` (the low-level client routine) must match.

If the user specifies the `RPC_ANYSOCK` argument, the RPC library routines will open sockets. Otherwise, they will expect the user to do so. The routines `svcudp_create()` and `clntudp_create()` will cause the RPC library routines to call `bind()` to bind their socket, if it is not bound already.

A service may choose to register its port number with the local portmapper service. This is done is done by specifying a nonzero protocol number in `svc_register()`. Incidently, a client can discover the server's port number by consulting the portmapper on the server's machine. This can be done automatically by specifying a zero port number in `clntudp_create()` or `clnttcp_create()`.

After creating an `SVCXPRT` pointer, the next step is to call `pmap_unset()` so that if the `nusers()` server crashed earlier, any previous trace of it is erased before restarting. More precisely, `pmap_unset()` erases the entry for `RUSERSPROG` from the port mapper's tables.

Finally, we associate the program number for `nusers()` with the procedure `nuser()`. The final argument to `svc_register()` is normally the protocol being used, which, in this case, is `IPPROTO_UDP`. Notice that unlike `registerrpc()`, there are no XDR routines involved in the registration process. Also, registration is done on the program level, rather than the procedure level.

The user routine `nuser()` must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by `nuser()` that `registerrpc()` handles automatically:

- Procedure `NULLPROC` (currently zero) returns without results. This can be used as a simple test for detecting if a remote program is running.
- There is a check for invalid procedure numbers. If one is detected, `svcerr_noproc()` is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via `svc_sendreply()`. Its first parameter is the `SVCXPRT` handle, the second is the XDR routine, and the third is a pointer to the data to be returned. As an example of how a server handles an RPC program that receives data, we can add a procedure `RUSERSPROC_BOOL` shown in Figure 77, which has an argument `nusers`, and returns `TRUE` or `FALSE` depending on whether there are `nusers` logged on.

Figure 77 Handling an RPC program that receives data

```
case RUSERSPROC_BOOL: {
    int bool;
    unsigned nuserquery;

    if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
       svcerr_decode(transp);
       return;
    }
    /*
     *  Code to set nusers = number of users
     */
    if (nuserquery == nusers)
       bool = TRUE;
    else bool = FALSE;

     if (!svc_sendreply(transp, xdr_bool, &bool)) {
       fprintf(stderr, "can't reply to RPC call\n");
       return (1);
    }

    return;

}
```

The relevant routine is `svc_getargs()`, which takes an `SVCXPRT` handle, the XDR routine, and a pointer to where the input is to be placed as arguments.

Memory allocation with XDR

XDR routines can allocate memory via malloc on your behalf, if you wish. This is why the second parameter of `xdr_array()` is a pointer to an array, rather than the array itself. If it is NULL, then `xdr_array()` allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider the following XDR routine `xdr_chararr1()`, which deals with a fixed array of bytes with length `SIZE`:

```
xdr_chararr1(xdrsp, chararr)
   XDR *xdrsp;
   char chararr[];

{
   char *p;
   int len;
   p = chararr;
   len = SIZE;
   return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

If space has already been allocated in `chararr[]`, it can be called from a server like this:

```
char chararr[SIZE];
svc_getargs(transp, xdr_chararr1, chararr);
```

If you want XDR to do the allocation, you must rewrite this routine as follows:

```
xdr_chararr2(xdrsp, chararrp)
   XDR *xdrsp;
   char **chararrp;
{
   int len;
   len = SIZE;
{
   return (xdr_bytes(xdrsp, charrarrp, &len,
            SIZE));
}
```

Then the RPC call will look something like this:

```
char *arrptr;
arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * Use the result here
 */

svc_freeargs(transp, xdr_chararr2, &arrptr);
```

Note that after being used, the character array can be freed with `svc_freeargs()`. `svc_freeargs()` will not attempt to free any memory if the pointer to it has a value of `NULL`. For example, in the routine `xdr_finalexample()` in Figure 75, if `finalp->string` is `NULL`, then it will not be freed. The same is true for `finalp->simplep`.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from `callrpc()`, the serializer is used. When called from `svc_getargs()`, the deserializer is used. And, when called from `svc_freeargs()`, the memory deallocator is used. When building simple examples like those in this section, a user doesn't have to worry about the three modes. Refer to "XDR Standard: Technical Notes" on page 289 for examples of more sophisticated XDR routines that determine their own mode and adjust their behavior accordingly.

The calling side

When you use `callrpc()`, you have no control over the RPC delivery mechanism or the socket used to transport the data. Figure 78 shows how to adjust these parameters in the code to call the `nusers` service:

Figure 78 Selecting a transport mechanism in an RPC call

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char **argv;

{
    struct hostent *hp;
    struct timeval pertry_timeout,
       total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    unsigned long nusers;

    if (argc != 2) {
       fprintf(stderr, "usage: nusers hostname\n");
       exit(-1);
    }

    if ((hp = gethostbyname(argv[1])) == NULL) {
       fprintf(stderr, "can't get addr for %s\n",argv[1]);
       exit(-1);
    }

    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
       hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;

    if ((client = clntudp_create(&server_addr, RUSERSPROG,
        RUSERSVERS, pertry_timeout, &sock)) == NULL) {
       clnt_pcreateerror("clntudp_create");
       exit(-1);
    }
```

Figure 78 Selecting a transport mechanism in an RPC call **(continued)**

```
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
        0, xdr_u_long, &nusers, total_timeout);

    if (clnt_stat != RPC_SUCCESS) {
    clnt_perror(client, "RPC");
    exit(-1);
    }

    clnt_destroy(client);
    close(sock);
    exit(0);
}
```

The low-level version of `callrpc()` is `clnt_call()` which takes a CLIENT pointer rather than a host name. The parameters to `clnt_call()` are a CLIENT pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the return value, a pointer to where the return value will be placed, and the time in seconds to wait for a reply.

The CLIENT pointer is encoded with the transport mechanism. `callrpc()` uses UDP, so it calls `clntudp_create()` to get a CLIENT pointer. To get TCP (Transmission Control Protocol), use `clnttcp_create()`.

The parameters to `clntudp_create()` are the server address, the program number, the version number, a timeout value (between tries), and a pointer to a socket. The final argument to `clnt_call()` is the total time to wait for a response. Therefore, the number of tries is the `clnt_call()` timeout divided by the `clntudp_create()` timeout.

Note that the `clnt_destroy()` call always deallocates the space associated with the CLIENT handle. It closes the socket associated with the CLIENT handle, however, only if the RPC library opened it. If the socket was opened by the user, it stays open. This makes it possible, in cases where there are multiple client handles using the same socket, to destroy one handle without closing the socket that other handles are using.

To make a stream connection, the call to `clntudp_create()` is replaced with a call to `clnttcp_create()`:

```
clnttcp_create(&server_addr, prognum,
   versnum, &sock, inputsize, outputsize);
```

There is no timeout argument; instead, the receive and send buffer sizes must be specified. When the `clnttcp_create()` call is made, a TCP connection is established. All RPC calls using that CLIENT handle would use this connection. The server side of an RPC call using TCP has `svcudp_create()` replaced by `svctcp_create()`:

```
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
```

The last two arguments to `svctcp_create()` are send and receive sizes respectively. If "0" is specified for either of these, the system chooses a reasonable default.

Other RPC features

This section discusses some other aspects of RPC that are occasionally useful.

Select on the server side

Suppose a process is processing RPC requests while performing some other activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling `svc_run()`. But if the other activity involves waiting on a a file descriptor, the `svc_run()` call won't work. The code for `svc_run()` is shown in Figure 79.

Figure 79 Select on the server side

```
void
svc_run()
{
        fd_set readfds;
        int dtbsz = getdtablesize();

        for (;;) {
           readfds = svc_fds;
           switch (select(dtbsz, &readfds, NULL,NULL,NULL)) {
              case -1:
                 if (errno == EINTR)
                      continue;
                 perror("select");
                 return;
              case 0:
                 break;
           default:
              svc_getreqset(&readfds);
           }
        }
}
```

You can bypass `svc_run()` and call `svc_getreqset()` yourself. All you need to know are the file descriptors of the socket(s) associated with the programs you are waiting on. Thus you can have your own `select()` that waits on both the RPC socket, and your own descriptors. Note that `svc_fds()` is a bit mask of all the file descriptors that RPC is using for services. It can change every time that any RPC library routine is called, because descriptors are constantly being opened and closed, for example for TCP connections.

Broadcast RPC

The portmapper is a daemon that converts RPC program numbers into DARPA protocol port numbers (refer to the portmap man page for details). You can't do broadcast RPC without the portmapper. The main differences between broadcast RPC and normal RPC calls are:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).

- Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UPD/IP.
- The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
- All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.
- Broadcast requests are limited in size to 1400 bytes. Replies can be up to 8800 bytes (the current maximum UDP packet size).

The broadcast RPC synopsis is:

```
#include <RPC/pmap_clnt.h>
. . .
enum clnt_stat     clnt_stat;
. . .
clnt_stat = clnt_broadcast(prognum, versnum, procnum, inproc, in,
    outproc, out,eachresult)
u_long       prognum;          /* program number */
u_long       versnum;          /* version number */
u_long       procnum;          /* procedure number */
xdrproc_t    inproc;           /* XDR routine for args */
caddr_t      in;               /* pointer to args */
xdrproc_t    outproc;          /* XDR routine for results */
caddr_t      out;              /* pointer to results */
bool_t       (*eachresult)(); /* call with each result */
```

The user provided procedure `eachresult()` is called each time a valid result is obtained. It returns a boolean that indicates whether or not the user wants more responses.

```
bool_t done;
. . .
    done = eachresult(resultsp, raddr)

caddr_t resultsp;
struct sockaddr_in *raddr;     /* Addr of responding machine */
```

If done is `TRUE`, then broadcasting stops and `clnt_broadcast()` returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with `RPC_TIMEDOUT`.

Batching

The RPC architecture is designed so that clients send a call message and wait for servers to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. This is inefficient if the client does not want or need an acknowledgment for every message sent. It is possible for clients to continue computing while waiting for a response, using RPC batch facilities.

RPC messages can be placed in a "pipeline" of calls to a desired server; this is called batching. Batching assumes

- Each RPC call in the pipeline requires no response from the server, and the server does not send a response message.
- The pipeline of calls is transported on a reliable byte stream transport such as TCP/IP.

Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Furthermore, the TCP/IP implementation can buffer up many call messages, and send them to the server in one write() system call. This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes, and the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually do a nonbatched call in order to flush the pipeline.

A contrived example of batching is shown in Figure 80. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service uses the TCP/IP transport.

Figure 80 Batching example

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <suntool/windows.h>

void windowdispatch();

main()
{
    SVCXPRT *transp;

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);

    if (transp == NULL){
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }

    pmap_unset(WINDOWPROG, WINDOWVERS);

    if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
        windowdispatch, IPPROTO_TCP)) {
        fprintf(stderr, "can't register WINDOW service\n");
        exit(1);
    }

    svc_run();  /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}

void
windowdispatch(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    char *s = NULL;

    switch (rqstp->rq_proc) {
        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can't reply to RPC call\n");
            return;
        case RENDERSTRING:
            if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                fprintf(stderr, "can't decode arguments\n");
```

Figure 80 Batching example **(continued)**

```
            /*
             *  Inform caller of error
             */
            svcerr_decode(transp);
            break;
        }

            /*
             *  Code here to render the string s
             */
            if (!svc_sendreply(transp, xdr_void, NULL))
                fprintf(stderr, "can't reply to RPC call\n");
            break;
        case RENDERSTRING_BATCHED:
            if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                fprintf(stderr, "can't decode arguments\n");

                /*
                 *  We are silent in the face of
                 *  protocol errors
                 */
                break;
            }

            /*
             *  Code here to render string s, but send
             *  no reply!
             */
            break;
        default:
            svcerr_noproc(transp);
            return;
    }

    /*
     *  Now free string allocated while decoding arguments
     */
    svc_freeargs(transp, xdr_wrapstring, &s);
}
```

The service could have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

In order for a client to take advantage of batching, the client must perform RPC calls on a TCP-based transport and the actual calls must have the following attributes:

- The result's XDR routine must be NULL
- The RPC call's timeout must be zero

Figure 81 shows an example of a client that uses batching to render several strings. The batching is flushed when the client gets a null string (EOF).

Figure 81 Using batching to render several strings

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <suntool/windows.h>

main(argc, argv)
    int argc;
    char **argv;

{
    struct hostent *hp;
    struct timeval pertry_timeout,
    total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    char buf[1000], *s = buf;

    if ((client = clnttcp_create(&server_addr,
    WINDOWPROG, WINDOWVERS, &sock, 0, 0)) == NULL) {
        perror("clnttcp_create");
        exit(-1);
    }
```

Figure 81 Using batching to render several strings **(continued)**

```
    total_timeout.tv_sec = 0;
    total_timeout.tv_usec = 0;

    while (scanf("%s", s) != EOF) {
        clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
            xdr_wrapstring, &s, NULL, NULL, total_timeout);

        if (clnt_stat != RPC_SUCCESS) {
            clnt_perror(client, "batched RPC");
            exit(-1);
        }
    }
    /* Now flush the pipeline */
    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
        xdr_void, NULL, total_timeout);

    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "RPC");
        exit(-1);
    }

    clnt_destroy(client);
    exit(0);
}
```

Because the server sends no message, clients cannot be notified of any failures that may occur and are responsible for handling errors.

The example in Figure 81 was completed to render all 2000 lines in the /etc/termcap file. The rendering service did nothing but throw the lines away. The example was run in the following four configurations with the results shown in Table 24:

Table 24 Comparison of regular to batched RPC execution times

Configuration	Result
Machine to itself, regular RPC	50 seconds
Machine to itself, batched RPC	16 seconds
Machine to another, regular RPC	52 seconds
Machine to another, batched RPC	10 seconds

Running `fscanf()` on /etc/termcap only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, although they are often difficult to design.

Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network file system, require stronger security than what has been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type is type none.

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support.

UNIX authentication—client side

When a caller creates a new RPC client handle as in:

```
clnt = clntudp_create(address, prognum,
   versnum, wait, sockp)
```

the appropriate transport instance defaults the associated authentication handle to be

```
clnt->cl_auth = authnone_create();
```

The RPC client can choose to use UNIX style authentication by setting `clnt->cl_auth` after creating the RPC client handle:

```
clnt->cl_auth = authunix_create_default();
```

This causes each RPC call associated with `clnt` to carry with it the following authentication credentials structure:

```
/*
 * UNIX style credentials.
 */
struct authunix_parms {
    u_long aup_time;          /* credentials creation time */
    char   *aup_machname;     /* host name where client is */
    int    aup_uid;           /* client's effective uid */
    int    aup_gid;           /* client's current group id */
    u_int  aup_len;           /* element length of aup_gids */
    int    *aup_gids;         /* array of groups user is in */
};
```

These fields are set by `authunix_create_default()` by invoking the appropriate system calls. Since the RPC user created this new style of authentication, the user is responsible for destroying it with:

```
auth_destroy(clnt->cl_auth);
```

This should be done in all cases, to conserve memory.

UNIX authentication—server side

Service implementors have a harder time dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine:

```
/*
 * An RPC Service request
 */
struct svc_req {
    u_long    rq_prog;              /* service program number */
    u_long    rq_vers;              /* service protocol version number */
    u_long    rq_proc;              /* desired procedure number */
    /*
     *  raw credentials from wire
     */
    struct opaque_auth rq_cred;
    caddr_t rq_clntcred;            /* credentials (read only) */
};
```

The `rq_cred` is mostly opaque, except for one field of interest—the style or flavor of authentication credentials:

```
/*
 * Authentication info. Mostly opaque to the programmer. */
 *
struct opaque_auth {
    enum_t  oa_flavor;          /* style of credentials */
    caddr_t oa_base;            /* address of more auth stuff */
    u_int   oa_length;          /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package guarantees the following to the service dispatch routine:

- That the request's `rq_cred` is well formed. Thus the service implementor may inspect the request's `rq_cred.oa_flavor` to determine which style of authentication the caller used. The service implementor may also wish to inspect the other fields of `rq_cred` if the style is not one of the styles supported by the RPC package.

- That the request's `rq_clntcred` field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. Remember that only unix style is currently supported, so (currently) `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is NULL the service implementor may wish to inspect the other (opaque) fields of `rq_cred` in case the service knows about a new type of authentication that the RPC package does not know about.

The remote users service example shown in Figure 74 and Figure 75 can be extended so that it computes results for all users except UID 16, as shown in Figure 82:

Figure 82 Remote users service example with UNIX authentication

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authunix_parms *unix_cred;
    int uid; unsigned long nusers;

    /*
     * we don't care about authentication for null proc
     */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
                return (1);
        }
         return;
    }

    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
        case AUTH_UNIX:
            unix_cred = (struct authunix_parms *)rqstp->rq_clntcred;
            uid = unix_cred->aup_uid;
            break;
        case AUTH_NULL:
        default:
            svcerr_weakauth(transp);
            return;
    }
```

Figure 82 Remote users service example with UNIX authentication **(continued)**

```
    switch (rqstp->rq_proc) {
        case RUSERSPROC_NUM:

            /*
             * make sure caller is allowed to call this proc
             */
            if (uid == 16) {
                svcerr_systemerr(transp);
                return;
            }

            /*
             * Code here to compute the number of users
             * and assign it to the variable nusers
             */
            if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
                fprintf(stderr, "can't reply to RPC call\n");
                return (1);
            }

            return;
        default:
            svcerr_noproc(transp);
            return;
    }
}
```

A few things should be noted here. First, it is customary not to check the authentication parameters associated with the `NULLPROC` (procedure number zero). Second, if the authentication parameter's type is not suitable for your service, you should call `svcerr_weakauth()`. And finally, the service protocol itself should return status for access denied. In the case of our example, the protocol does not have such a status, so we call the service primitive `svcerr_systemerr()` instead.

The last point underscores the relation between the RPC authentication package and the services. RPC deals only with authentication and not with individual services' access control; the services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

DES authentication

UNIX authentication is quite easy to defeat. Instead of using `authunix_create_default()`, you can call `authunix_create()` and then modify the RPC authentication handle. `authunix_create()` returns by furnishing the desired user ID and host name. DES authentication is recommended if you need more security than UNIX authentication offers.

The details of the DES authentication protocol are complicated and are not explained here. Refer to "Remote Procedure Calls: Protocol Specification" on page 345 for details.

For DES authentication to work, the keyserv(8c) daemon must be running on both server and client machines. Users on these machines must have public keys assigned by the network administrator in the publickey(5) database. They also must have decrypted their secret keys using their login password. This automatically happens when you log in with login(1). You can also do this manually using keylogin(1).

Client side

If a client wishes to use DES authentication, it must set its authentication handle appropriately. For example:

```
cl->cl_auth = authdes_create(servername, 60,
   &server_addr, NULL);
```

The first argument is the network name of the owner of the server process. Typically, server processes are root processes, and their network name can be derived using the following call:

```
char servername[MAXNETNAMELEN];

host2netname(servername, rhostname, NULL);
```

Here, `rhostname` is the host name of the machine the server process is running on. `host2netname()` furnishes `servername` to contain this root process's network name. If the server process was run by a regular user, one could use the call `user2netname()` instead. Here is an example for a server process with the same user ID as the client.

```
char servername[MAXNETNAMELEN];

user2netname(servername, getuid(), NULL);
```

The last argument to both `user2netname()` and `host2netname()` is the name of the NIS domain where the server is located. The `NULL` used here means "use the local default domain name."

The second argument to `authdes_create()` is a lifetime for the credential. Here it is set to sixty seconds. This means that the credential will expire 60 seconds from when authdes_create() is called. If a user tries to reuse the credential, the server RPC subsystem will recognize that it has expired and not grant any requests. If the same user tries to reuse the credential within the sixty second lifetime, they will still be rejected because the server RPC subsystem remembers which credentials it has already seen in the near past, and will not grant requests to duplicates.

The third argument to `authdes_create()` is the address of the host with which to synchronize. In order for DES authentication to work, the server and client must agree on the time. Here the address of the server itself is passed, so the client and server will both be using the server's time. A `NULL` argument indicates that further synchronization is not necessary. You should only do this if you are sure the client and server are already synchronized.

The final argument to `authdes_create()` is the address of a DES encryption key to use for encrypting timestamps and data. If this argument is `NULL`, as it is in this example, a random key will be chosen. The client may find out the encryption key being used by consulting the `ah_key` field of the authentication handle.

Server Side

The server side is simpler than the client side. The example in Figure 82 has been rewritten as shown in Figure 83 to use AUTH_DES instead of AUTH_UNIX.

Figure 83 Remote users service example with DES authentication

```
#include <sys/time.h>
#include <RPC/auth_des.h>
. . .
nuser(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
    struct authdes_cred *des_cred;
    int uid;
    int gid;
    int gidlen;
    int gidlist[10];

    /*
     * we don't care about authentication for null proc
     */
    if (rqstp->rq_proc == NULLPROC) { /* same as before */
    }

    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
       case AUTH_DES:
          des_cred = (struct authdes_cred *) rqstp->rq_clntcred;

          if (! netname2user(des_cred->adc_fullname.name,
              &uid, &gid, &gidlen, gidlist)) {
              fprintf(stderr, "unknown user: %s\n",
                  des_cred->adc_fullname.name);
              svcerr_systemerr(transp);
              return;
          }
          break;
       case AUTH_NULL:
       default:
          svcerr_weakauth(transp);
          return;
    }
    /*
     * The rest is the same as before
     */
```

Note

The routine `netname2user()` is the inverse of `user2netname()`. It takes a network ID and converts it to a unix ID. `netname2user()` also supplies group IDs which are not used in this example, but may be useful in other ConvexOS programs.

Using `inetd()`

An RPC server can be started from `inetd()`. The only difference from the usual code is that the service creation routine should be called in the following form since `inetd()` passes a socket as file descriptor 0:

```
transp = svcudp_create(0);          /* For UDP */
transp = svctcp_create(0,0,0);      /* For listener TCP sockets*/
transp = svcfd_create(0,0,0);       /* For connected TCP sockets*/
```

Also, `svc_register()` should be called as

```
svc_register(transp, PROGNUM, VERSNUM,
   service, 0);
```

with the final flag as 0, because the program will already be registered by `inetd()`. Remember, if you want to exit from the server process and return control to `inetd()`, you need to explicitly exit, since `svc_run()` never returns.

The format of entries in /etc/inetd.conf for RPC services is in one of the following two forms:

service_name `dgram udp wait|nowait` *user server_program* [*args*]

service_name `stream tcp nowait` *user server_program* [*args*]

where

service_name	Symbolic name of the service as it appears in the /etc/rpc file.
`dgram` or `stream`	Socket type.
udp or tcp	Protocol—`tcp` for streams sockets and `udp` for datagram sockets.
`wait` or `nowait`	Flag indicating whether the server processes all incoming messages on the socket (`wait`) or frees the socket after receiving a message (`nowait`). stream sockets should always specify `nowait`.
user	User name under which the server runs.
server_program	Path name of the program inetd executes when it receives a request on the socket.
args	Arguments passed to the server program.

For RPC, the *server_program* has a version number specified in front of the path name for the program. If the same program handles multiple versions, the version number can be a range, as in this example:

```
rup dgram udp wait root 1-3 /usr/etc/rpc.rstatd rstatd
```

For more information on `inetd`, refer to the inetd man page.

More examples

This section contains some additional RPC programming examples using version numbers, the TCP protocol, and callback procedures.

Versions

By convention, the first version number of program PROG is PROGVERS_ORIG, and the most recent version is PROGVERS. Suppose there is a new version of the user program that returns an unsigned short rather than a long. If we name this version RUSERSVERS_SHORT, then a server that wants to support both versions can double register, as shown below.

```
if (!svc_register(transp, RUSERSPROG,
   RUSERSVERS_ORIG, nuser, IPPROTO_TCP)) {
   fprintf(stderr, "can't register RUSER
      service\n");
   exit(1);
}

if (!svc_register(transp, RUSERSPROG,
   RUSERSVERS_SHORT, nuser, IPPROTO_TCP)) {
   fprintf(stderr, "can't register RUSER
      service\n");
   exit(1);
}
```

Both versions can be handled by the same C procedure, as in the example in Figure 84.

Figure 84 Handling multiple program versions

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2;

    switch (rqstp->rq_proc) {

        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "can't reply to RPC call\n");
                return (1);
            }
            return;

        case RUSERSPROC_NUM:
            /*
             * Code here to compute the number of users
             * and assign it to the variable nusers
             */
            nusers2 = nusers;

            switch (rqstp->rq_vers) {
                case RUSERSVERS_ORIG:
                    if (!svc_sendreply(transp, xdr_u_long,&nusers)){
                        fprintf(stderr, "can't reply to RPC call\n");
                    }
                    break;
                case RUSERSVERS_SHORT:
                    if (!svc_sendreply(transp, xdr_u_short,&nusers2)){
                        fprintf(stderr, "can't reply to RPC call\n");
                    }
                    break;
            }

        default:
            svcerr_noproc(transp);
            return;
        }
    }
```

TCP

Here is an example that is essentially RPC. The initiator of the `rpcsnd()` call takes its standard input and sends it to the server `rcv` which prints it on standard output. The RPC call uses TCP. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

Figure 85 RCP call using TCP

```
/*
 * The XDR routine:
 * on decode, read from wire, write onto fp
 * on encode, read from fp, write onto wire
 */
#include <stdio.h>
#include <rpc/rpc.h>

xdr_rpc(xdrs, fp)
    XDR *xdrs;
    FILE *fp;

{
    unsigned long size;
    char buf[BUFSIZ], *p;

    if (xdrs->x_op == XDR_FREE) /* nothing to free */
        return 1;

    while (1) {
        if (xdrs->x_op == XDR_ENCODE) {
            if ((size = fread(buf,sizeof(char),
                BUFSIZ,fp)) == 0
                && ferror(fp)) {
                fprintf(stderr, "can't fread\n");
                return (1);
            }
        }
        p = buf;

        if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
            return 0;

        if (size == 0)
            return 1;
```

Figure 85 RCP call using TCP **(continued)**

```
        if (xdrs->x_op == XDR_DECODE) {
            if (fwrite(buf, sizeof(char), size, fp) != size) {
                fprintf(stderr, "can't fwrite\n");
                return (1);
            }
        }
    }
}
/*
 * The sender routines
 */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>

main(argc, argv)
    int argc;
    char **argv;

{
    int xdr_rpc();
    int err;

    if (argc < 2) {
        fprintf(stderr, "usage: %s servername\n", argv[0]);
        exit(-1);
    }

    if ((err = callrpctcp(argv[1], RPCPROG, RPCPROC, RPCVERS,
        xdr_rpc, stdin, xdr_void, 0) != 0)) {
        clnt_perrno(err);
        fprintf(stderr, "can't make RPC call\n");
        exit(1);
    }

    exit(0);
}

callrpctcp(host, prognum, procnum, versnum, inproc, in, outproc, out)
    char *host, *in, *out;
    xdrproc_t inproc, outproc;
```

Figure 85 RCP call using TCP **(continued)**

```
{
    struct sockaddr_in server_addr;
    int socket = RPC_ANYSOCK;
    enum clnt_stat clnt_stat;
    struct hostent *hp;
    register CLIENT *client;
    struct timeval total_timeout;

    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "can't get addr for '%s'\n", host);
        return (-1);
    }

    bcopy(hp->h_addr,(caddr_t)&server_addr.sin_addr,h->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;

    if ((client = clnttcp_create(&server_addr, prognum, versnum, &socket,
        BUFSIZ, BUFSIZ)) == NULL) {
        perror("rpctcp_create");
        return (-1);
    }

    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum, inproc, in, outproc, out,
        total_timeout);
    clnt_destroy(client);
    return (int)clnt_stat;
}

/*
 * The receiving routines
 */
#include <stdio.h>
#include <rpc/rpc.h>

main()
{
    register SVCXPRT *transp;
    int rpc_service(),
    xdr_rpc();

    if ((transp = svctcp_create(RPC_ANYSOCK, BUFSIZ, BUFSIZ)) == NULL) {
        fprintf("svctcp_create: error\n");
        exit(1);
    }
```

Figure 85 RCP call using TCP **(continued)**

```
    pmap_unset(RPCPROG, RPCVERS);

    if (!svc_register(transp, RPCPROG, RPCVERS, rpc_service, IPPROTO_TCP)){
        fprintf(stderr, "svc_register: error\n");
        exit(1);
    }

    svc_run();  /* never returns */
    fprintf(stderr, "svc_run should never return\n");
}

rpc_service(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
        case NULLPROC:
            if (svc_sendreply(transp, xdr_void, 0) == 0) {
                fprintf(stderr, "err: rpc_service");
                return (1);
            }

            return;
        case RPCPROC:

            if (!svc_getargs(transp, xdr_rpc, stdout)) {
                svcerr_decode(transp);
                return;
            }

            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "can't reply\n");
                return;
            }

            return (0);
        default:
            svcerr_noproc(transp);
            return;
    }
}
```

Callback procedures

Occasionally, it is useful to have a server become a client and make an RPC call back to the process that is its client. An example is remote debugging, where the client is a window system program, and the server is a debugger running on the remote machine. Most of the time, the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a break point, the roles are reversed, and the debugger wants to make an RPC call to the window program so that it can inform the user that a break point has been reached.

To do an RPC callback, you need a program number on which to make the RPC call. Since this will be a dynamically generated program number, it should be in the transient range

```
0x40000000 - 0x5fffffff
```

Figure 86 shows how to obtain a program number. The routine `gettransient()` returns a valid program number in the transient range, and registers it with the portmapper. It only talks to the portmapper running on the same machine as the `gettransient()` routine itself. The call to `pmap_set()` is a test and set operation in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it. On return, the `sockp` argument will contain a socket that can be used as the argument to an `svcudp_create()` or `svctcp_create()` call.

Figure 86 Obtaining a program number for RPC callback

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>

gettransient(proto, vers, sockp)
    int proto, vers, *sockp;
{
    static int prognum = 0x40000000;
    int s, len, socktype;
    struct sockaddr_in addr;

    switch(proto) {
        case IPPROTO_UDP:
            socktype = SOCK_DGRAM;
            break;
        case IPPROTO_TCP:
            socktype = SOCK_STREAM;
            break;
        default:
            fprintf(stderr, "unknown protocol type\n");
            return 0;
    }

    if (*sockp == RPC_ANYSOCK) {

        if ((s = socket(AF_INET, socktype, 0)) < 0) {
            perror("socket");
            return (0);
        }

        *sockp = s;
    }

    else s = *sockp;

    addr.sin_addr.s_addr = 0;
    addr.sin_family = AF_INET;
    addr.sin_port = 0;
    len = sizeof(addr);
```

Figure 86 Obtaining a program number for RPC callback **(continued)**

```
    /*
     * may be already bound, so don't check for error
     */
    bind(s, &addr, len);

    if (getsockname(s, &addr, &len)< 0) {
        perror("getsockname");
        return (0);
    }

    while (!pmap_set(prognum++, vers, proto,
        ntohs(addr.sin_port)))
        continue; return (prognum-1);

}
```

Note

The call to `ntohs()` is necessary to ensure that the port number in `addr.sin_port`, which is in network byte order, is passed in host byte order (as `pmap_set()` expects). See the byteorder(3N) man page for more details on the conversion of network addresses from network to host byte order.

Figure 87 and Figure 88 illustrate how to use the `gettransient()` routine. The client makes an RPC call to the server, passing it a transient program number. The client then waits to receive a callback from the server at that program number. The server registers the program `EXAMPLEPROG()` so that it can receive the RPC call informing it of the callback program number. At some random time (on receiving an `ALRM` signal in this example), it sends a callback RPC call, using the program number it received earlier.

Figure 87 RPC callback example—client side

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>

int callback();
char hostname[256];

main()
{
    int x, ans, s;
    SVCXPRT *xprt;

    gethostname(hostname, sizeof(hostname));
    s = RPC_ANYSOCK;
    x = gettransient(IPPROTO_UDP, 1, &s);
    fprintf(stderr, "client gets prognum %d\n", x);

    if ((xprt = svcudp_create(s)) == NULL) {
        fprintf(stderr, "rpc_server: svcudp_create\n");
        exit(1);
    }

    /* protocol is 0 - gettransient does registering */
    (void)svc_register(xprt, x, 1, callback, 0);
    ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
    EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);

    if ((enum clnt_stat) ans != RPC_SUCCESS) {
        fprintf(stderr, "call: ");
        clnt_perrno(ans);
        fprintf(stderr, "\n");
    }

    svc_run();
    fprintf(stderr, "Error: svc_run shouldn't return\n");
}

callback(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
```

Figure 87 RPC callback example—client side **(continued)**

```
{
    switch (rqstp->rq_proc) {
        case 0:
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: exampleprog\n");
                return (1);
            }
            return (0);
        case 1:
            if (!svc_getargs(transp, xdr_void, 0)) {
                svcerr_decode(transp);
                return (1);
            }
            fprintf(stderr, "client got callback\n");

            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: exampleprog");
                return (1);
            }
    }
}
```

Figure 88 RPC callback example—server side

```
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char *getnewprog();
char hostname[256];
int docallback();
int pnum; /* program number for callback routine */

main()
{
    gethostname(hostname, sizeof(hostname));
    registerrpc(EXAMPLEPROG, EXAMPLEVERS, EXAMPLEPROC_CALLBACK,
       getnewprog, xdr_int, xdr_void);
    fprintf(stderr, "server going into svc_run\n");
    signal(SIGALRM, docallback);
    alarm(10);
    svc_run();
    fprintf(stderr, "Error: svc_run shouldn't return\n");
}


char *
getnewprog(pnump)
    char *pnump;

{
    pnum = *(int *)pnump;
    return NULL;
}

docallback()
{
    int ans;

    ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0, xdr_void, 0);

    if (ans != 0) {
       fprintf(stderr, "server: ");
       clnt_perrno(ans);
       fprintf(stderr, "\n");
    }
}
```

XDR Standard: Technical Notes

17

This chapter contains technical notes on Sun's implementation of the External Data Representation (XDR) standard, a set of library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. For a formal specification of the XDR standard, refer to "XDR Standard: Protocol Specification" on page 371.

XDR is the backbone of Sun's Remote Procedure Call package in the sense that data for remote procedure calls is transmitted using the standard. XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine. For a compete specification of the system External Data Representation routines, see the xdr(3N) man pages.

This chapter contains a short tutorial overview of the XDR library routines, a guide to accessing currently available XDR streams, and information on defining new streams and data types. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) will only need the information in the sections "Number filters" on page 297, "Floating point filters" on page 298, and "Enumeration filters" on page 298. Programmers wishing to implement RPC and XDR on new machines will be interested in this chapter in its entirety, as well as "XDR Standard: Protocol Specification" on page 371, which is our primary reference to XDR.

Note

`rpcgen` can be used to write XDR routines even in cases where no RPC calls are being made.

On Sun systems, C programs that want to use XDR routines must include the file <rpc/rpc.h>, which contains all the necessary interfaces to the XDR system. Since the C library libc.a contains all the XDR routines, compile the C programs as usual by entering:

```
% cc program.c
```

where *program* is the C program you want to compile.

Justification

Consider the following two programs shown in Figure 89 and Figure 90, "writer" and "reader", respectively.

Figure 89 Writer program

```
#include <stdio.h>

main()        /* writer.c */
{
    long i;

    for (i = 0; i < 8; i++) {
        if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1){
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
    exit(0);
}
```

Figure 90 Reader program

```
#include <stdio.h>

main()    /* reader.c */
{
    long i, j;

    for (j = 0; j < 8; j++) {
        if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

These two programs appear to be portable, for the following reasons:

- They pass lint checking
- They exhibit the same behavior when executed on two different hardware architectures, a Sun and a VAX.

Piping the output of the writer program to the reader program gives identical results on a Sun or a VAX.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%

vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

With the advent of local area networks and 4.2BSD came the concept of "network pipes" — a process producing data on one machine, and a second process consuming data on another machine. You can construct a network pipe with writer and reader. Here are the results of producer and consumer programs. The first program produces data on a Sun, and the second program consumes data on a VAX.

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864
83886080 100663296 117440512
sun%
```

Identical results can be obtained by executing writer on the VAX and reader on the Sun. These results occur because the byte ordering of long integers differs between the VAX and the Sun, even though word size is the same. Note that the decimal value 16777216 is 2 to the 24th power—when four bytes are reversed, the 1 winds up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the `read()` and `write()` calls with calls to an XDR library routine `xdr_long()`, a filter that knows the standard representation of a long integer in its external form.

Figure 91 and Figure 92 show the revised versions of the writer and reader programs, respectively.:

Figure 91 Revised writer program

```
#include <stdio.h>
#include <rpc/rpc.h>        /* XDR is a sub-library of RPC */

main()                      /* writer.c */
{
    XDR xdrs;
    long i;

    xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
    for (i = 0; i < 8; i++) {
        if (!xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
    exit(0);
}
```

Figure 92 Revised reader program

```
#include <stdio.h>
#include <rpc/rpc.h>        /* XDR is a sub-library of RPC */

main()                      /* reader.c */
{
    XDR xdrs;
    long i, j;

    xdrstdio_create(&xdrs, stdin, XDR_DECODE);
    for (j = 0; j < 8; j++) {
        if (!xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

The new programs are executed on a Sun, on a VAX, and from a Sun to a VAX. The results are shown below.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%

vax% writer | reader
0 1 2 3 4 5 6 7
vax%

sun% writer | rsh vax reader
0 1 2 3 4 5 6 7
sun%
```

Note

Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. And pointers, which are very convenient to use, have no meaning outside the machine where they are defined.

A canonical standard

XDR's approach to standardizing data representations is canonical. That is, XDR defines a single byte order (Big Endian), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations. Similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents.

The single standard completely decouples programs that create or send portable data from those that use or receive portable data. The advent of a new machine or a new language has no effect upon the community of existing portable data creators and users. A new machine joins this community by being "taught" how to convert the standard representations and its local representations. The local representations of other machines are irrelevant. Conversely, to existing programs running on other machines, the local representations of the new machine are also irrelevant; such programs can immediately read portable data produced by the new machine because such data conforms to the canonical standards that they already understand.

There are strong precedents for XDR's canonical approach. For example, TCP/IP, UDP/IP, XNS, Ethernet, and, indeed, all protocols below layer five of the ISO model, are canonical protocols. The advantage of any canonical approach is simplicity. In the case of XDR, a single set of conversion routines is written once and is never touched again.

The canonical approach has a disadvantage that is unimportant in real-world data transfer applications. Suppose two Little-Endian machines are transferring integers according to the XDR standard. The sending machine converts the integers from Little-Endian byte order to XDR (Big-Endian) byte order. The receiving machine performs the reverse conversion. Because both machines observe the same byte order, their conversions are unnecessary. The point, however, is not necessity, but cost as compared to the alternative.

The time spent converting to and from a canonical representation is insignificant, especially in networking applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure. To transmit a tree, for example, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there. Storage for the leaf may have to be deallocated as well. Similarly, to receive a tree, storage must be allocated for each leaf, data must be moved from the buffer to the leaf and properly aligned, and pointers must be constructed to link the leaves together. Every machine pays the cost of traversing and copying data structures whether or not conversion is required.

In networking applications, communications overhead—the time required to move the data down through the sender's protocol layers, across the network and up through the receiver's protocol layers—dwarfs conversion overhead.

The XDR library

The XDR library not only solves data portability problems, it also allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it can make sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements or pointers to other structures.

Let's examine the two programs more closely. There is a family of XDR stream creation routines in which each member treats the stream of bits differently. In our example, data is manipulated using standard I/O routines, so we use `xdrstdio_create`. The parameters to XDR stream creation routines vary according to their function. In our example, `xdrstdio_create()` takes a pointer to an XDR structure that it initializes, a pointer to a FILE on which the input or output is performed, and the operation. The operation may be XDR_ENCODE for serializing in the writer program, or XDR_DECODE for deserializing in the reader program.

Note

RPC users never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the users.

The `xdr_long()` primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns FALSE (0) if it fails, and TRUE (1) if it succeeds. Second, for each data type, *xxx*, there is an associated XDR routine of the form:

```
xdr_xxx(xdrs, xp)
    XDR *xdrs;
    xxx *xp;
{
}
```

In our case, *xxx* is long, and the corresponding XDR routine is a primitive, `xdr_long()` The client can also define an arbitrary structure *xxx* in which case the client will supply the routine xdr_*xxx* describing each field by calling XDR routines of the appropriate type. In all cases the first parameter xdrs can be treated as an opaque handle and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation—this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself—only in the case of deserialization is the object modified.

This feature is not shown in our trivial example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, the user can obtain the direction of the XDR operation. Refer to the section "XDR operation directions" on page 310 for details.

Let's look at a slightly more complicated example. Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type:

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
};
```

The corresponding XDR routine describing this structure is as follows:

```
bool_t   /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    if (xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities))
        return(TRUE);
    return(FALSE);
}
```

Note

The parameter `xdrs` is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call and to give up immediately and return FALSE if the subroutine fails.

This example also shows that the type `bool_t` is declared as an integer whose only values are TRUE (1) and FALSE (0). This document uses the following definitions:

```
#define bool_t     int
#define TRUE       1
#define FALSE      0
```

Keeping these conventions in mind, `xdr_gnumbers()` can be rewritten as:

```
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    return(xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

XDR library primitives

This section gives a synopsis of each XDR primitive. It starts with basic data types and moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file <rpc/xdr.h> automatically included by <rpc/rpc.h>.

Number filters

The XDR library provides primitives to translate between numbers and their corresponding external representations. Primitives cover the set of numbers in:

```
[signed, unsigned] * [short, int, long]
```

Specifically, the eight primitives are:

```
bool_t xdr_char(xdrs, cp)
   XDR *xdrs;
   char *cp;

bool_t xdr_u_char(xdrs, ucp)
   XDR *xdrs;
   unsigned char *ucp;

bool_t xdr_int(xdrs, ip)
   XDR *xdrs;
   int *ip;

bool_t xdr_u_int(xdrs, up)
   XDR *xdrs;
   unsigned *up;

bool_t xdr_long(xdrs, lip)
   XDR *xdrs;
   long *lip;

bool_t xdr_u_long(xdrs, lup)
   XDR *xdrs;
   u_long *lup;

bool_t xdr_short(xdrs, sip)
   XDR *xdrs;
   short *sip;

bool_t xdr_u_short(xdrs, sup)
   XDR *xdrs;
   u_short *sup;
```

The first parameter, `xdrs`, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE if they do not.

Floating point filters

The XDR library also provides primitive routines for C's floating point types:

```
bool_t xdr_float(xdrs, fp)
   XDR *xdrs;
   float *fp;

bool_t xdr_double(xdrs, dp)
   XDR *xdrs;
   double *dp;
```

The first parameter, `xdrs`, is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. Both routines return `TRUE` if they complete successfully, and `FALSE` if they do not.

Note

Since the numbers are represented in IEEE floating point, routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice-versa.

Enumeration filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C type of enum has the same representation inside the machine as a C type of integer. The boolean type is an important instance of the enum. The external representation of a boolean is always `TRUE` (`1`) or `FALSE` (`0`).

```
#define bool_t    int
#define FALSE     0
#define TRUE      1
#define enum_t    int

bool_t xdr_enum(xdrs, ep)
   XDR *xdrs;
   enum_t *ep;

bool_t xdr_bool(xdrs, bp)
   XDR *xdrs;
   bool_t *bp;
```

The second parameters, `ep` and `bp`, are addresses of the associated type that provides data to, or receives data from, the stream `xdrs`.

No data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();  /* always returns TRUE */
```

Constructed data type filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives previously discussed. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may utilize memory management. In many cases, memory is allocated when deserializing data with `XDR_DECODE`. Therefore, the XDR package must provide means to deallocate memory. This is done by an XDR operation, `XDR_FREE`. To review, the three XDR directional operations are `XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`.

Strings

In C, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a char * and not a sequence of characters. The external representation of a string is drastically different from its internal representation. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers.

The `xdr_string` routine converts between the two representations of strings.

```
bool_t xdr_string(xdrs, sp, maxlength)
   XDR *xdrs;
   char **sp;
   u_int maxlength;
```

The first parameter, `xdrs`, is the XDR stream handle. The second parameter, `sp`, is a pointer to a string type `char **`. The third parameter, `maxlength`, specifies the maximum number of bytes allowed during encoding or decoding. Its value is usually specified by a protocol. For example, a protocol specification may say that a file name can be no longer than 255 characters.

The routine returns FALSE if the number of characters exceeds maxlength and TRUE if it doesn't.

Caution

Keep maxlength small. If it is too big you can blow the heap, since xdr_string() will call malloc() for space.

The behavior of xdr_string() is similar to the behavior of other routines discussed in this section. The direction XDR_ENCODE is easiest to understand. The parameter sp points to a string of a certain length. If the string does not exceed maxlength, the bytes are serialized.

The effect of deserializing a string is subtle. First, the length of the incoming string is determined. (It must not exceed maxlength.) Next, sp is dereferenced. If the value is NULL, then a string of the appropriate length is allocated and *sp is set to this string. If the original value of *sp is non-null, then the XDR package assumes that a target area has been allocated, which can hold strings no longer than maxlength. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the XDR_FREE operation, the string is obtained by dereferencing sp. If the string isn't NULL, it is freed, and *sp is set to NULL. In this operation, xdr_string() ignores the maxlength parameter.

Byte arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways:

- The length of the array (the byte count) is explicitly located in an unsigned integer.
- The byte sequence is not terminated by a null character.
- The external representation of the bytes is the same as their internal representation.

The primitive xdr_bytes() converts between the internal and external representations of byte arrays:

```
bool_t
   xdr_bytes(xdrs, bpp, lp, maxlength)
   XDR *xdrs;
   char **bpp;
   u_int *lp;
   u_int maxlength;
```

The usage of the first, second, and fourth parameters is identical to the first, second and third parameters of `xdr_string`, respectively. The length of the byte area is obtained by dereferencing lp when serializing. `*lp` is set to the byte length when deserializing.

Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. The `xdr_bytes()` routine treats a subset of generic arrays in which the size of array elements is known to be 1, and the external description of each element is built-in. The generic array primitive, `xdr_array()` requires parameters identical to those of `xdr_bytes()`, plus two more: the size of array elements, and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```
bool_t
xdr_array(xdrs, ap, lp, maxlength,
          elementsiz, xdr_element)
   XDR *xdrs;
   char **ap;
   u_int *lp;
   u_int maxlength;
   u_int elementsiz;
   bool_t (*xdr_element)();
```

The parameter `ap` is the address of the pointer to the array. If `*ap` is NULL when the array is being deserialized, XDR allocates an array of the appropriate size and sets `*ap` to that array. The element count of the array is obtained from `*lp` when the array is serialized. `*lp` is set to the array length when the array is deserialized. The parameter maxlength is the maximum number of elements that the array is allowed to have. `elementsiz` is the byte size of each element of the array. The C function `sizeof()` can be used to obtain this value. The `xdr_element()` routine is called to serialize, deserialize, or free each element of the array.

Before defining more constructed data types, it is appropriate to present three examples.

Identifying a network user

A user on a networked machine can be identified by:

- Machine name, such as krypton (see the gethostname man page)
- User's UID (see the geteuid man page)

- Group numbers to which the user belongs (see the getgroups man page)

A structure with this information and its associated XDR routine could be coded as shown in Figure 93.

Figure 93 Identifying a user on the network

```
struct netuser {
    char *nu_machinename;
    int   nu_uid;
    u_int nu_glen;
    int   *nu_gids;
};

#define NLEN 255     /* machine names < 256 chars */
#define NGRPS 20     /* user can't be in > 20 groups */

bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
{
    return(xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
       xdr_int(xdrs, &nup->nu_uid) &&
       xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen,
       NGRPS, sizeof (int), xdr_int));
}
```

Identifying a party of network users

A party of network users could be implemented as an array of netuser structure. The declaration and its associated XDR routines are shown in Figure 94.

Figure 94 Identifying a party of network users

```
struct party {
    u_int p_len;
    struct netuser *p_nusers;
};

#define PLEN 500    /* max number of users in a party */

bool_t
    xdr_party(xdrs, pp)
    XDR *xdrs;
    struct party *pp;
{
    return(xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
        sizeof (struct netuser), xdr_netuser));
}
```

Writing a command structure

The well-known parameters to main, `argc` and `argv`, can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like the code shown in Figure 95.

Figure 95 Command structure

```
struct cmd {
    u_int c_argc;
    char **c_argv;
};

#define ALEN 1000    /* args cannot be > 1000 chars */
#define NARGC 100    /* commands cannot have > 100 args */

struct history {
    u_int h_len;
    struct cmd *h_cmds;
};

#define NCMDS 75     /* history is no more than 75 commands */

bool_t
    xdr_wrap_string(xdrs, sp)
    XDR *xdrs;
    char **sp;
{
    return(xdr_string(xdrs, sp, ALEN));
}

bool_t xdr_cmd(xdrs, cp)
    XDR *xdrs;
    struct cmd *cp;
{
    return(xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
        sizeof (char *), xdr_wrap_string));
}

bool_t
    xdr_history(xdrs, hp)
    XDR *xdrs;
    struct history *hp;
{
    return(xdr_array(xdrs, &hp->h_cmds, &hp->h_len,
        NCMDS,sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine `xdr_wrap_string()` is needed to package the `xdr_string()` routine. This is because the implementation of `xdr_array()` passes only two parameters to the array element description routine. `xdr_wrap_string()` supplies the third parameter to `xdr_string()`.

By now the recursive nature of the XDR library should be obvious. Let's continue with more constructed data types.

Opaque data

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say that handles are opaque. The `xdr_opaque()` primitive is used for describing fixed sized, opaque bytes.

```
bool_t
xdr_opaque(xdrs, p, len)
   XDR *xdrs;
   char *p;     /* the location of the bytes */
   u_int len;   /* the number of bytes in the
                 * opaque object */
```

By definition, the actual data contained in the opaque object are not machine portable.

Fixed sized arrays

The XDR library provides a primitive, `xdr_vector()` for fixed-length arrays.

```
#define NLEN 255      /* machine names must be
                       * 256 chars */
#define NGRPS 20      /* user belongs to
                       *exactly 20 groups */

struct netuser {
   char *nu_machinename;
   int nu_uid;
   int nu_gids[NGRPS];
};

bool_t
xdr_netuser(xdrs, nup)
   XDR *xdrs;
   struct netuser *nup;
{
   int i;

   if (!xdr_string(xdrs,
      &nup->nu_machinename, NLEN))
      return(FALSE);

   if (!xdr_int(xdrs, &nup->nu_uid))
```

```
        return(FALSE);

    if (!xdr_vector(xdrs, nup->nu_gids,
        NGRPS, sizeof(int), xdr_int)) {
        return(FALSE);
    }
    return(TRUE);
}
```

Discriminated unions

The XDR library supports discriminated unions. A discriminated union is a C union and an `enum_t` value that selects an "arm" of the union.

```
struct xdr_discrim {
    enum_t value;
    bool_t (*proc)();
};

bool_t
    xdr_union(xdrs, dscmp, unp, arms,
        defaultarm)
    XDR *xdrs;
    enum_t *dscmp;
    char *unp;
    struct xdr_discrim *arms;
    bool_t (*defaultarm)(); /* may equal
                             * NULL */
```

First, the routine translates the discriminant of the union located at `*dscmp`. The discriminant is always of type `enum_t`. Next, the union located at `*unp` is translated. The parameter arms is a pointer to an array of `xdr_discrim` structures. Each structure contains an ordered pair of [`value`, `proc`]. If the union's discriminant is equal to the associated value, then the procedure is called to translate the union. The end of the `xdr_discrim` structure array is denoted by a routine of value `NULL` (0). If the discriminant is not found in the arms array, then the default arm procedure is called, when it is non-null; otherwise the routine returns `FALSE`.

Suppose the type of a union can be integer, character pointer (a string), or a `gnumbers` structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };
struct u_tag {
   enum utype utype;    /* the union's
                         * discriminant */
   union {
      int ival;
      char *pval;
      struct gnumbers gn;
   }
   uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
   { INTEGER, xdr_int },
   { GNUMBERS, xdr_gnumbers },
   { STRING, xdr_wrap_string },
   /* always terminate arms with a NULL
    * xdr_proc */
   { __dontcare__, NULL }
}
bool_t
xdr_u_tag(xdrs, utp)
   XDR *xdrs;
   struct u_tag *utp;
{
   return(xdr_union(xdrs, &utp->utype,
      &utp->uval, u_tag_arms, NULL));
}
```

The routine `xdr_gnumbers()` was presented previously in the XDR Library section. `xdr_wrap_string()` was presented in example C. The default arm parameter to `xdr_union()` (the last parameter) is NULL in this example. Therefore, the value of the union's discriminant may legally take on only values listed in the `u_tag_arms` array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

It is worth pointing out that the values of the discriminant may be sparse, though, in this example, they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

As an exercise, implement `xdr_union()` using the other primitives in this section.

Pointers

In C it is often convenient to put pointers to another structure within a structure. The `xdr_reference()` primitive makes it easy to serialize, deserialize, and free these referenced structures.

```
bool_t
xdr_reference(xdrs, pp, size, proc)
   XDR *xdrs;
   char **pp;      /* address of the pointer
                    * to the structure */
   u_int ssize;    /* size in bytes of the
                    * structure */
   bool_t (*proc)(); /* XDR routine that
                      * describes the
                      * structure */
```

To obtain the value of the parameter `ssize`, use the C function `sizeof()`. When decoding data, storage is allocated if `*pp` is `NULL`.

There is no need for a primitive `xdr_struct()` to describe structures within structures because pointers are always sufficient.

As an exercise, implement `xdr_reference()` using `xdr_array`.

Caution

`xdr_reference()` and `xdr_array()` are NOT interchangeable external representations of data.

Suppose there is a structure containing a person's name and a pointer to a `gnumbers` structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
   char *name;
   struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is shown in Figure 96.

Figure 96 Pointer to a structure within a structure

```
bool_t
xdr_pgn(xdrs, pp)
    XDR *xdrs;
    struct pgn *pp;
{
    if (xdr_string(xdrs, &pp->name, NLEN) &&
        xdr_reference(xdrs, &pp->gnp,
        sizeof(struct gnumbers), xdr_gnumbers))
        return(TRUE);
    return(FALSE);
}
```

Pointer semantics and XDR

In many applications, C programmers attach double meaning to the values of a pointer. Typically, the value `NULL` (or zero) indicates data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For instance, in Figure 96, a `NULL` pointer value for `gnp` could indicate that the person's assets and liabilities are unknown. That is, the pointer value encodes two things: whether or not the data is known; and, if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive `xdr_reference()` doesn't attach any special meaning to a null-value pointer during serialization. That is, passing an address of a pointer whose value is `NULL` to `xdr_reference()` when serialing data will most likely cause a memory fault and, depending on your system, a core dump.

`xdr_pointer()` correctly handles `NULL` pointers. For more information about its use, refer to the section "Linked lists" on page 316.

As an exercise, after reading the section "Linked lists" on page 316, return here and extend the example in Figure 96 so that it can correctly handle NULL pointer values.

For additional practice using the `xdr_union`, `xdr_reference()`, and `xdr_void()` primitives, implement a generic pointer handling primitive that implicitly handles `NULL` pointers. (Implement `xdr_pointer`.)

Nonfilter primitives

XDR streams can be manipulated with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
   XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
   XDR *xdrs;
   u_int pos;

xdr_destroy(xdrs)
   XDR *xdrs;
```

The routine `xdr_getpos()` returns an unsigned integer that describes the current position in the data stream.

Caution

In some XDR streams, the returned value of `xdr_getpos()` is meaningless. The routine returns a –1 in this case (even though –1 should be a legitimate value).

The routine `xdr_setpos()` sets a stream position to `pos`.

Caution

In some XDR streams, setting a position is impossible. In such cases, `xdr_setpos()` will return `FALSE`. This routine will also fail if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The `xdr_destroy()` primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

XDR operation directions

At times you may wish to optimize XDR routines by taking advantage of the direction of the operation — `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`. The value `xdrs->x_op` always contains the direction of the XDR operation. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in section "Linked lists" on page 316 demonstrates the usefulness of the `xdrs->x_op` field.

XDR stream access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream.

Streams currently exist for (de)serialization of data to or from standard I/O FILE streams, TCP/IP connections and ConvexOS files, and memory.

Standard I/O streams

XDR streams can be interfaced to standard I/O using the `xdrstdio_create()` routine as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>     /* XDR streams part
                          * of RPC */

void
   xdrstdio_create(xdrs, fp, x_op)
   XDR *xdrs;
   FILE *fp;
   enum xdr_op x_op;
```

The routine `xdrstdio_create()` initializes an XDR stream pointed to by `xdrs`. The XDR stream interfaces to the standard I/O library. Parameter `fp` is an open file, and `x_op` is an XDR direction.

Memory streams

Memory streams allow the streaming of data into or out of a specified area of memory:

```
#include <rpc/rpc.h>

void
xdrmem_create(xdrs, addr, len, x_op)
   XDR *xdrs;
   char *addr;     /* pointer to local memory
                    * address */
u_int len;         /* length of memory, in
                    * bytes */
enum xdr_op x_op;
```

The routine `xdrmem_create()` initializes an XDR stream in local memory. The parameters `xdrs` and `x_op` are identical to the corresponding parameters of `xdrstdio_create()`. Currently, the UDP/IP implementation of RPC uses `xdrmem_create()`. Complete call, or result messages, is built in memory before calling the `sendto()` system routine.

Record (TCP/IP) streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the ConvexOS file or 4.2 BSD connection interface.

```
#include <rpc/rpc.h>     /* XDR streams part
                          * of RPC */
xdrrec_create(xdrs,sendsize, recvsize,
      iohandle, readproc, writeproc)
   XDR *xdrs;
   u_int sendsize, recvsize;
   char *iohandle;
   int (*readproc)(), (*writeproc)();
```

The routine `xdrrec_create()` provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal ConvexOS files.

The parameter `xdrs` is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters sendsize and recvsize determine the size in bytes of the output and input buffers, respectively. If their values are zero (0), then predetermined defaults are used.

When a buffer needs to be filled or flushed, the routine `readproc()` or `writeproc()` is called, respectively. The usage and behavior of these routines are similar to the ConvexOS system calls `read()` and `write()`. However, the first parameter to each of these routines is the opaque parameter iohandle. The other two parameters `buf` and `nbytes` and the results (byte count) are identical to the system routines. If *xxx* is `readproc()` or `writeproc()`, then it has the following form:

```
/*
 * returns the actual number of bytes
 * transferred. -1 is an error
 */
int
xxx(iohandle, buf, len)
   char *iohandle;
   char *buf;
   int nbytes;
```

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in the section "Advanced topics" on page 316. The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
   XDR *xdrs;
   bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
   XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
   XDR *xdrs;
```

The routine `xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is `TRUE`, then the stream's `writeproc()` will be called; otherwise, `writeproc()` will be called when the output buffer has been filled.

The routine `xdrrec_skiprecord()` causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, then the routine `xdrrec_eof()` returns `TRUE`. (There is no more data in the underlying file descriptor.)

XDR stream implementation

This section describes the abstract data types needed to implement new instances of XDR streams.

The XDR object

The structure shown in Figure 97 defines the interface to an XDR stream.

Figure 97 Interface to an XDR stream

```
enum xdr_op { XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2 };

typedef struct {
    enum xdr_op x_op;           /* operation;fast added param */
    struct xdr_ops {
        bool_t (*x_getlong)();  /* get long from stream */
        bool_t (*x_putlong)();  /* put long to stream */
        bool_t (*x_getbytes)(); /*get bytes from stream */
        bool_t (*x_putbytes)(); /* put bytes to stream */
        u_int (*x_getpostn)();  /* return stream offset */
        bool_t (*x_setpostn)(); /* reposition offset */
        caddr_t (*x_inline)();  /* ptr to buffered data */
        VOID (*x_destroy)();    /* free private area */
    } *x_ops;
    caddr_t x_public;           /* users' data */
    caddr_t x_private;          /* pointer to private data */
    caddr_t x_base;             /* private for position info */
    int x_handy;                /* extra private word */
} XDR;
```

The `x_op` field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value.

The fields `x_private`, `x_base`, and `x_handy` are private to the particular stream's implementation. The field `x_public` is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives. `x_getpostn()`, `x_setpostn()`, and `x_destroy()` are macros for accessing operations.

The operation `x_inline()` takes two parameters: an XDR * and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment have been consumed or put. The routine may return `NULL` if it cannot return a buffer segment of the requested size. (The `x_inline()` routine is for cycle squeezers. Use of the resulting buffer is not data-portable. Users are encouraged not to use this feature.)

The operations `x_getbytes()` and `x_putbytes()` blindly get and put sequences of bytes from or to the underlying stream. They return `TRUE` if they are successful, and `FALSE` if they are not. The routines have identical parameters (replace *xxx*).

```
bool_t
xxxbytes(xdrs, buf, bytecount)
    XDR *xdrs;
    char *buf;
    u_int bytecount;
```

The operations `x_getlong()` and `x_putlong()` receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The primitives `htonl()` and `ntohl()` can be helpful in accomplishing this. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return `TRUE` if they succeed, and `FALSE` otherwise. They have identical parameters:

```
bool_t
xxxlong(xdrs, lp)
    XDR *xdrs;
    long *lp;
```

Implementors of new XDR streams must make an XDR structure (with new operation routines) available to clients, using some kind of create routine.

Advanced topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier sections, the following examples are written using both the XDR C library routines and the XDR data description language. The chapter "XDR Standard: Protocol Specification" on page 371 describes this language in complete detail.

Linked lists

The example in the section "The XDR library" on page 294 presented a C data structure and its associated XDR routines for a individual's gross assets and liabilities. The example is duplicated below:

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
};

bool_t
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    if (xdr_long(xdrs, &(gp->g_assets)))
        return(xdr_long(xdrs,
            &(gp->g_liabilities)));
    return(FALSE);
}
```

Now assume that we wish to implement a linked list of such information. A data structure can be constructed as follows:

```
struct gnumbers_node {
    struct gnumbers gn_numbers;
    struct gnumbers_node *gn_next;
};

typedef struct gnumbers_node *gnumbers_list;
```

The head of the linked list can be thought of as the data object. That is, the head is not merely a convenient shorthand for a structure. Similarly the `gn_next` field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the `gn_next` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive declaration of `gnumbers_list`:

```
struct gnumbers {
    int g_assets;
    int g_liabilities;
};

struct gnumbers_node {
    gnumbers gn_numbers;
    gnumbers_node *gn_next;
};
```

In the description shown in Figure 98, the boolean indicates whether there is more data following it. If the boolean is `FALSE`, then it is the last data field of the structure. If it is `TRUE`, then it is followed by a `gnumbers` structure and (recursively) by a `gnumbers_list`.

Note

The C declaration has no boolean explicitly declared in it (even though the `gn_next` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing the XDR routines for a `gnumbers_list` follow easily from the XDR description above. The primitive `xdr_pointer()` is used to implement the XDR union above.

Figure 98 Linked list example (recursive)

```
bool_t
xdr_gnumbers_node(xdrs, gn)
    XDR *xdrs;
    gnumbers_node *gn;
{
    return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
       xdr_gnumbers_list(xdrs, &gp->gn_next));
}

bool_t
xdr_gnumbers_list(xdrs, gnp)
    XDR *xdrs;
    gnumbers_list *gnp;
{
    return(xdr_pointer(xdrs, gnp,
       sizeof(struct gnumbers_node),
       xdr_gnumbers_node));
}
```

The unfortunate side effect of using XDR on a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to recursion.

The routine shown in Figure 99 collapses the previous two mutually recursive routines into a single, nonrecursive one.

Figure 99 Linked list example (nonrecursive)

```
bool_t
xdr_gnumbers_list(xdrs, gnp)
    XDR *xdrs;
    gnumbers_list *gnp;
{
    bool_t more_data;
    gnumbers_list *nextp;

    for (;;) {
       more_data = (*gnp != NULL);
       if (!xdr_bool(xdrs, &more_data)) {
          return(FALSE);
       }
       if (! more_data) {
          break;
       }
       if (xdrs->x_op == XDR_FREE) {
          nextp = &(*gnp)->gn_next;
       {
       if (!xdr_reference(xdrs, gnp,
          sizeof(struct gnumbers_node), xdr_gnumbers)) {

       return(FALSE);
       }
       gnp = (xdrs->x_op == XDR_FREE) ?
          nextp : &(*gnp)->gn_next;
    }
    *gnp = NULL;
    return(TRUE);
}
```

The first task is to find out whether there is more data or not, so that the boolean information can be serialized. Notice that this statement is unnecessary in the XDR_DECODE case, since the value of more_data is not known until we deserialize it in the next statement.

The next statement uses XDR for the more_data field of the XDR union. Then, if there is truly no more data, we set this last pointer to NULL to indicate the end of the list and return TRUE indicating we are done. Note that setting the pointer to NULL is only important in the XDR_DECODE case, since it is already NULL in the XDR_ENCODE and XDR_FREE cases.

Next, if the direction is XDR_FREE, we set the value of nextp to indicate the location of the next pointer in the list. We do this at this point because we need to dereference gnp to find the location of the next item in the list. After the next statement, the storage to which gnp points will be freed and will no longer be valid. We can't do this for all directions, for in the XDR_DECODE direction, the value of gnp won't be set until the next statement.

Next, we XDR the data in the node using the primitive xdr_reference(). xdr_reference() is like xdr_pointer(), which we used before, but it does not send over the boolean indicating whether there is more data. We use it instead of xdr_pointer() because we have already referenced this information ourselves using XDR. Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is xdr_gnumbers for referencing gnumbers using XDR, but each element in the list is actually of type gnumbers_node. We don't pass xdr_gnumbers_node() because it is recursive, and, instead, we use xdr_gnumbers(), which references all of the nonrecursive part using XDR. Note that this trick will work only if the gn_numbers field is the first item in each element, so that their addresses are identical when passed to xdr_reference.

Finally, we update gnp to point to the next item in the list. If the direction is XDR_FREE, we set it to the previously saved value, otherwise we can dereference gnp to get the proper value. Even though it is harder to understand than the recursive version, this nonrecursive routine is far less likely to blow the C stack. It will also run more efficiently since a lot of procedure call overhead has been removed. Most lists are small (in the hundreds of items or less) and the recursive version should be sufficient for them.

Network File System: Version 2 Protocol Specification

18

This chapter contains the protocol specification for Network File System Version 2, designated RFC 1094 by the ARPA-Internet Network Information Center.

NFS protocol definition

Servers have been known to change over time, as well as the protocol that they use; so, RPC provides a version number with each RPC request. This RFC describes Version 2 of the NFS protocol. Even in the Version 2, there are various obsolete procedures and parameters that will be removed in later versions.

File system model

NFS assumes a file system that is hierarchical, with directories as all but the bottom-level files. Each entry in a directory (file, directory, device, etc.) has a string name. Different operating systems may have restrictions on the depth of the tree or the names used, as well as using different syntax to represent the "pathname", which is the concatenation of all the "components" (directory and file names) in the name. A "file system" is a tree on a single server (usually a single disk or physical partition) with a specified "root." Some operating systems provide a "mount" operation to make all file systems appear as a single tree, while others maintain a "forest" of file systems. Files are unstructured streams of uninterpreted bytes.

NFS looks up one component of a pathname at a time. It may not be obvious why it does not just take the whole pathname, traipse down the directories, and return a file handle when it is done. There are several good reasons not to do this. First, pathnames need separators between the directory components, and different

operating systems use different separators. We could define a Network Standard Pathname Representation, but then every pathname would have to be parsed and converted at each end. Other issues are discussed in "NFS implementation issues" on page 338.

Although files and directories are similar objects in many ways, different procedures are used to read directories and files. This provides a network standard format for representing directories. The same argument as above could have been used to justify a procedure that returns only one directory entry per call. The problem is efficiency. Directories can contain many entries, and a remote call to return each would be just too slow.

RPC information

Authentication:

The NFS service uses `AUTH_UNIX()`, `AUTH_DES()`, or `AUTH_SHORT()` style authentication, except in the `NULL` procedure where `AUTH_NONE()` is also allowed.

Port Protocols:

NFS currently is supported on UDP/IP only.

Port Number:

The NFS protocol currently uses the UDP port number 2049. This is not an officially assigned port, so later versions of the protocol use the portmapping facility of RPC.

Sizes of XDR structures

These are the sizes, given in decimal bytes, of various XDR structures used in the protocol:

```
/*
 * The maximum number of bytes of data in a
 * READ or WRITE request
 */
const MAXDATA = 8192;

/* The maximum number of bytes in a pathname
 * argument
 */
const MAXPATHLEN = 1024;

/* The maximum number of bytes in a file name
 * argument
```

```
 */
const MAXNAMLEN = 255;

/*
 * The size in bytes of the opaque "cookie"
 * passed by READDIR
 */
const COOKIESIZE = 4;

/* The size in bytes of the opaque file
 * handle
 */
const FHSIZE = 32;
```

Basic data types

The following XDR definitions are basic structures and types used in other structures described further on.

stat

```
enum stat {
   NFS_OK = 0,
   NFSERR_PERM=1,
   NFSERR_NOENT=2,
   NFSERR_IO=5,
   NFSERR_NXIO=6,
   NFSERR_ACCES=13,
   NFSERR_EXIST=17,
   NFSERR_NODEV=19,
   NFSERR_NOTDIR=20,
   NFSERR_ISDIR=21,
   NFSERR_FBIG=27,
   NFSERR_NOSPC=28,
   NFSERR_ROFS=30,
   NFSERR_NAMETOOLONG=63,
   NFSERR_NOTEMPTY=66,
   NFSERR_DQUOT=69,
   NFSERR_STALE=70,
   NFSERR_WFLUSH=99
};
```

The `stat` type is returned with every procedure's results. A value of `NFS_OK` indicates that the call completed successfully and the results are valid. Other values indicate some kind of error occurred on the server side during the servicing of the procedure. The following error values are derived from UNIX error numbers:

NFSERR_PERM
Not owner. The caller does not have correct ownership to perform the requested operation.

NFSERR_NOENT
No such file or directory. The file or directory specified does not exist.

NFSERR_IO
Some sort of hard error occurred when the operation was in progress. This could be a disk error, for example.

NFSERR_NXIO
No such device or address.

NFSERR_ACCES
Permission denied. The caller does not have the correct permission to perform the requested operation.

NFSERR_EXIST
File exists. The file specified already exists.

NFSERR_NODEV
No such device.

NFSERR_NOTDIR
Not a directory. The caller specified a non-directory in a directory operation.

NFSERR_ISDIR
Is a directory. The caller specified a directory in a non- directory operation.

NFSERR_FBIG
File too large. The operation caused a file to grow beyond the server's limit.

NFSERR_NOSPC
No space left on device. The operation caused the server's file system to reach its limit.

NFSERR_ROFS
Read-only file system. Write attempted on a read-only file system.

NFSERR_NAMETOOLONG
File name too long. The file name in an operation was too long.

NFSERR_NOTEMPTY
Directory not empty. Attempted to remove a directory that was not empty.

NFSERR_DQUOT
Disk quota exceeded. The client's disk quota on the server has been exceeded.

NFSERR_STALE
The fhandle given in the arguments was invalid. That is, the file referred to by that file handle no longer exists, or access to it has been revoked.

NFSERR_WFLUSH
The server's write cache used in the WRITECACHE() call got flushed to disk.

ftype

```
enum ftype {    /* enumeration ftype gives the type of a
                 * file */
   NFNON = 0,   /* indicates a non-file */
   NFREG = 1,   /* a regular file */
   NFDIR = 2,   /* a directory */
   NFBLK = 3,   /* a block-special device */
   NFCHR = 4,   /* a character-special device */
   NFLNK = 5    /* a symbolic link */
};
```

fhandle

```
typedef opaque fhandle[FHSIZE];
```

The fhandle is the file handle passed between the server and the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

timeval

```
struct timeval {
   unsigned int seconds;
   unsigned int useconds;
};
```

The timeval structure is the number of seconds and microseconds since midnight January 1, 1970, Greenwich Mean Time. It is used to pass time and date information.

```
struct fattr {                  /* attributes of a file */
    ftype type;                 /* type of the file */
    unsigned int mode;          /* access mode encoded as a set of bits */
    unsigned int nlink;         /* number of hard links to the file */
    unsigned int uid;           /* user identification number of the owner
                                 * of the file */
    unsigned int gid;           /* group identification number of the group
                                 * of the file */
    unsigned int size;          /* size in bytes of the file */
    unsigned int blocksize;     /* size in bytes of a block of the file */
    unsigned int rdev;          /* device number of the file */
    unsigned int blocks;        /* if rdev is type NFCHR or NFSBLK, blocks
                                 * is the number of blocks the file takes
                                 * up on disk */
    unsigned int fsid;          /* file system identifier for the file
                                 * system containing the file */
    unsigned int fileid;        /* number that uniquely identifies the file
                                 * within its file system */
    timeval atime;              /* time when the file was last accessed for
                                 * either read or write */
    timeval mtime;              /* time when the file data was last
                                 * modified (written) */
    timeval ctime;              /* time when the status of the file was
                                 * last changed */
};
```

nlink also indicates the number of different names for the same file. Writing to the file also changes ctime if the size of the file changes.

The descriptions given below specify the bit positions using octal numbers.

Table 25 Bit positions

Bit	Description
0040000	This is a directory; `type` field should be NFDIR.
0020000	This is a character special file; `type` field should be NFCHR.
0060000	This is a block special file; `type` field should be NFBLK.
0100000	This is a regular file; `type` field should be NFREG.
0120000	This is a symbolic link file; `type` field should be NFLNK.
0140000	This is a named socket; `type` field should be NFNON.
0004000	Set user id on execution.
0002000	Set group id on execution.
0001000	Save swapped text even after use.
0000400	Read permission for owner.
0000200	Write permission for owner.
0000100	Execute and search permission for owner.
0000040	Read permission for group.
0000020	Write permission for group.
0000010	Execute and search permission for group.
0000004	Read permission for others.
0000002	Write permission for others.
0000001	Execute and search permission for others.

The bits are the same as the mode bits returned by the stat(2) system call. The file type is specified both in the mode bits and in the file type. This is fixed in future versions.

The `rdev` field in the attributes structure is an operating system specific device specifier.

sattr

```
struct sattr {
   unsigned int mode:
   unsigned int uid;
   unsigned int gid;
   unsigned int size;
   timeval atime;
   timeval mtime;
};
```

The `sattr` structure contains the file attributes which can be set from the client. The fields are the same as for fattr previously described. A size of zero indicates the file will be truncated. A value of -1 indicates a field that will be ignored.

filename

```
typedef string filename<MAXNAMLEN>;
```

The type `filename` is used for passing file names or pathname components.

path

```
typedef string path<MAXPATHLEN>;
```

The type `path` is a pathname. The server considers it as a string without internal structure, but to the client it is the name of a node in a file system tree.

attrstat

```
struct attrstat switch (stat status) {
   case NFS_OK:
      fattr attributes;
   default:
      void;
};
```

The `attrstat` structure is a common procedure result. It contains a status and, if the call succeeds, it also contains the attributes of the file on which the operation was done.

diropres

```
union diropres switch (stat status) {
   case NFS_OK:
      struct {
         fhandle file; /* new file handle */
/*
 * attributes associated with the new file
 * handle
 */
         fattr attributes;
      } diropok;
   default:
         void;
};
```

The results of a directory operation are returned in a `diropres` structure. If the call succeeds, a new file handle file and the `fattr` attributes are returned along with the status.

Server procedures

The protocol definition is given as a set of procedures with arguments and results defined using the RPC language. A brief description of the function of each procedure should provide enough information to allow implementation.

All of the procedures in the NFS protocol are assumed to be synchronous. When a procedure returns to the client, the client can assume that the operation has completed and any data associated with the request is now on stable storage. For example, a client `WRITE` request may cause the server to update data blocks, file system information blocks (such as indirect blocks), and file attribute information (size and modify times). When the `WRITE` returns to the client, it can assume that the write is safe, even in case of a server crash, and it can discard the data written. This is a very important part of the statelessness of the server. If the server would wait to flush data from remote requests, the client would have to save those requests so that it could resend them in case of a server crash.

```
/* Remote file service routines */
program NFS_PROGRAM {
   version NFS_VERSION {
      void NFSPROC_NULL(void) = 0;
      attrstat NFSPROC_GETATTR(fhandle) = 1;
      attrstat NFSPROC_SETATTR(sattrargs) = 2;
      void NFSPROC_ROOT(void) = 3;
      diropres NFSPROC_LOOKUP(diropargs) = 4;
      readlinkres NFSPROC_READLINK(fhandle) = 5;
      readres NFSPROC_READ(readargs) = 6;
      void NFSPROC_WRITECACHE(void) = 7;
      attrstat NFSPROC_WRITE(writeargs) = 8;
      diropres NFSPROC_CREATE(createargs) = 9;
      stat NFSPROC_REMOVE(diropargs) = 10;
      stat NFSPROC_RENAME(renameargs) = 11;
      stat NFSPROC_LINK(linkargs) = 12;
      stat NFSPROC_SYMLINK(symlinkargs) = 13;
      diropres NFSPROC_MKDIR(createargs) = 14;
      stat NFSPROC_RMDIR(diropargs) = 15;
      readdirres NFSPROC_READDIR(readdirargs)
         = 16;
      statfsres NFSPROC_STATFS(fhandle) = 17;
   } = 2;
} = 100003;
```

Do nothing

```
void
NFSPROC_NULL(void) = 0;
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

Get file attributes

```
attrstat
NFSPROC_GETATTR (fhandle) = 1;
```

If the reply status is `NFS_OK`, then the reply attributes contain the attributes for the file given by the input `fhandle`.

Set file attributes

```
struct sattrargs {
   fhandle file;
   sattr attributes;
   };

attrstat
NFSPROC_SETATTR (sattrargs) = 2;
```

The attributes argument contains fields which are either -1 or are the new value for the attributes of the fhandle file. If the reply status is NFS_OK, then the reply attributes have the attributes of the file after the SETATTR operation completes.

Note

The use of -1 to indicate an unused field in attributes is changed in the next version of the protocol.

Get file system root

```
void
NFSPROC_ROOT(void) = 3;
```

Obsolete. This procedure is no longer used because finding the root file handle of a file system requires moving pathnames between client and server. To do this correctly, we would need to define a network standard representation of pathnames. Instead, the function of looking up the root file handle is done by the MNTPROC_MNT() procedure. (Refer to the section "Mount protocol definition" on page 340 for details.)

Look up file name

```
diropres
NFSPROC_LOOKUP(dirop args) = 4;
```

If the reply status equals NFS_OK, then the reply `file` and reply `attributes` are the file handle and attributes for the file `name` in the directory given by `dir` in the argument.

Read from symbolic link

```
union readlinkres switch (stat status) {
   case NFS_OK:
      path data;
   default:
      void;
};

readlinkres
NFSPROC_READLINK(fhandle) = 5;
```

If `status` has the value `NFS_OK`, then the reply `data` is the data in the symbolic link given by the file referred to by the fhandle argument.

Note

Since NFS always parses pathnames on the client, the pathname in a symbolic link may mean something different (or be meaningless) on a different client or on the server if a different pathname syntax is used.

Read from file

```
struct readargs {
   fhandle file;
   unsigned offset;
   unsigned count;
   unsigned totalcount;
};

union readres switch (stat status) {
   case NFS_OK:
      fattr attributes;
      opaque data<NFS_MAXDATA>;
   default:
      void;
};

readres
NFSPROC_READ(readargs) = 6;
```

Returns up to `count` bytes of `data` from the file given by `file`, starting at `offset` bytes from the beginning of the file. The first byte of the file is at offset zero. The file attributes after the read takes place are returned in `fattr` structure `attributes`.

Note

The argument `totalcount` is unused and is removed in the next protocol revision.

Write to cache

```
void
NFSPROC_WRITECACHE(void) = 7;
```

Write to file

```
struct writeargs {
   fhandle file;
   unsigned beginoffset;
   unsigned offset;
   unsigned totalcount;
   opaque data<NFS_MAXDATA>;
};

attrstat
NFSPROC_WRITE(writeargs) = 8;
```

Writes `data` beginning `offset` bytes from the beginning of `file`. The first byte of the file is at offset zero. If the reply `status` is `NFS_OK`, then the reply `attributes` contains the attributes of the file after the write is finished. The write operation is atomic. Data from this call to `WRITE` will not be mixed with data from another client's calls.

Note

The arguments `beginoffset` and `totalcount` are ignored and are removed in the next protocol revision.

Create file

```
struct createargs {
   diropargs where;
   sattr attributes;
};

diropres
NFSPROC_CREATE(createargs) = 9;
```

The file `name` is created in the directory given by `dir`. The initial attributes of the new file are given by `attributes`. A reply status of `NFS_OK` indicates that the file was created, and the replies `file` and `attributes` are its file handle and attributes, respectively. Any other reply status means that the operation failed, and no file was created.

Note

This routine should pass an exclusive create flag, meaning "create the file only if it is not already there". This will be added in the next protocol revision.

Remove file

```
stat
NFSPROC_REMOVE(diropargs) = 10;
```

The file `name` is removed from the directory given by `dir`. A reply of `NFS_OK` means the directory entry was removed.

Note

Possibly non-idempotent operation. A non-idempotent operation interferes with operations with other clients by retrying unsuccessful calls.

Rename file

```
struct renameargs {
   diropargs from;
   diropargs to;
};

stat
NFSPROC_RENAME(renameargs) = 11;
```

The existing file `from.name` in the directory given by `from.dir` is renamed to to.name in the directory given by `to.dir`. If the reply is `NFS_OK`, the file was renamed. The `RENAME` operation is atomic on the server; it cannot be interrupted.

Note

Possibly non-idempotent operation.

Create link to file

```
struct linkargs {
   fhandle from;
   diropargs to;
};

stat
FSPROC_LINK(linkargs) = 12;
```

Creates the file `to.name` in the directory given by `to.dir`, which is a hard link to the existing file given by `from`. If the return value is `NFS_OK`, a link was created. Any other return value indicates an error, and the link was not created.

A hard link should have the property that changes to either of the linked files are reflected in both files. When a hard link is made to a file, the attributes for the file should have a value for `nlink` that is one greater than the value before the link.

Note

Possibly non-idempotent operation.

Create symbolic link

```
struct symlinkargs {
   dirоpargs from;
   path to;
   sattr attributes;
};

stat
NFSPROC_SYMLINK(symlinkargs) = 13;
```

Creates the file `from.name` with ftype `NFLNK` in the directory given by `from.dir`. The new file contains the pathname `to` and has initial attributes given by `attributes`. If the return value is `NFS_OK`, a link was created. Any other return value indicates an error, and the link was not created.

A symbolic link is a pointer to another file. The name given in `to` is not interpreted by the server, only stored in the newly created file. When the client references a file that is a symbolic link, the contents of the symbolic link are normally transparently reinterpreted as a pathname to substitute. A `READLINK` operation returns the data to the client for interpretation.

Note

On CONVEX servers the attributes are never used, since symbolic links always have mode 0777.

Create directory

```
diropres
NFSPROC_MKDIR (createargs) = 14;
```

The new directory `where.name` is created in the directory given by `where.dir`. The initial attributes of the new directory are given by `attributes`. A reply status of `NFS_OK` indicates that the new directory was created, and replies `file` and `attributes` are its file handle and attributes, respectively. Any other reply status means that the operation failed and no directory was created.

Note

Possibly non-idempotent operation.

Remove directory

```
stat
NFSPROC_RMDIR(dirоpargs) = 15;
```

The existing empty directory `name` in the directory given by `dir` is removed. If the reply is `NFS_OK`, the directory was removed.

Note

Possibly non-idempotent operation.

Read from directory

```
struct readdirargs {
   fhandle dir;
   nfscookie cookie;
   unsigned count;
};

struct entry {
   unsigned fileid;
   filename name;
   nfscookie cookie;
   entry *nextentry;
};

union readdirres switch (stat status) {
   case NFS_OK:
      struct {
         entry *entries;
         bool eof;
      } readdirok;
   default:
      void;
};

readdirres
NFSPROC_READDIR (readdirargs) = 16;
```

Returns a variable number of directory entries, with a total size of up to `count` bytes, from the directory given by `dir`. If the returned value of `status` is `NFS_OK`, then it is followed by a variable number of `entry` entries. Each `entry` contains a `fileid` which consists of a unique number to identify the file within a file system, the name of the file, and a `cookie`, which is an opaque pointer to the next entry in the directory. The `cookie` is used in the next `READDIR` call to get more entries starting at a given point in the directory. The special `cookie` zero (all bits zero) can be used to get the entries starting at the beginning of the directory. The `fileid` field should be the same number as the fileid in the attributes of the file. (Refer to the section "Basic data types" on page 323.) The `eof` flag has a value of `TRUE`, if there are no more entries in the directory.

Get file system attributes

```
union statfsres (stat status) {
     case NFS_OK:
      struct {
         unsigned tsize;
         unsigned bsize;
         unsigned blocks;
         unsigned bfree;
       unsigned bavail;
       } info;
  default:
       void;
};

statfsres
NFSPROC_STATFS(fhandle) = 17;
```

If the reply status is `NFS_OK`, then the reply info gives the attributes for the file system that contains file referred to by the input `fhandle`. The attribute fields contain the following values:

`tsize`—The optimum transfer size of the server in bytes. This is the number of bytes the server would like to have in the data part of READ and WRITE requests.

`bsize`—The block size in bytes of the file system.

blocks—The total number of bsize blocks on the file system.

`bfree`—The number of free bsize blocks on the file system.

`bavail`—The number of bsize blocks available to non-privileged users.

Note

This call does not work well if a file system has variable size blocks.

NFS implementation issues

The NFS protocol is designed to be operating system independent, but many operations have semantics similar to the operations of the UNIX file system. This section discusses some of the implementation-specific semantic issues.

Server/client relationship

The NFS protocol is designed to allow servers to be as simple and general as possible. Sometimes the simplicity of the server can be a problem, if the client wants to implement complicated file system semantics.

For example, some operating systems allow removal of open files. A process can open a file and, while it is open, remove it from the directory. The file can be read and written as long as the process keeps it open, even though the file has no name in the file system. It is impossible for a stateless server to implement these semantics. The client can do some tricks such as renaming the file on remove, and only removing it on close. The server provides enough functionality to implement most file system semantics on the client.

Every NFS client can also potentially be a server, and remote and local mounted file systems can be freely intermixed. This leads to some interesting problems when a client travels down the directory tree of a remote file system and reaches the mount point on the server for another remote file system. Allowing the server to follow the second remote mount would require loop detection, server lookup, and user revalidation. Instead, clients are not permitted to cross a server's mount point. When a client does a LOOKUP on a directory on which the server has mounted a file system, the client sees the underlying directory instead of the mounted directory. A client can do remote mounts that match the server's mount points to maintain the server's view.

Pathname interpretation

There are a few complications to the rule that pathnames are always parsed on the client. For example, symbolic links can have different interpretations on different clients. Another common problem for some implementations is the special interpretation of the pathname "..", which indicates the parent of a given directory.

Permission issues

The NFS protocol, strictly speaking, does not define the permission checking used by servers. However, it is expected that a server will do normal operating system permission checking using `AUTH_UNIX` style authentication as the basis of its protection mechanism. The server gets the client's effective "uid," effective "gid," and groups on each call and uses them to check permission. There are various problems with this method that can be resolved in interesting ways.

Using "uid" and "gid" implies that the client and server share the same "uid" list. Every server and client pair must have the same mapping from user to "uid" and from group to "gid." Since every client can also be a server, this tends to imply that the whole network shares the same "uid/gid" space. `AUTH_DES` (and the next revision of the NFS protocol) uses string names instead of numbers, but there are still complex problems to be solved.

Another problem arises due to the usually stateful open operation. Most operating systems check permission at open time, and then check that the file is open on each read and write request. With stateless servers, the server has no idea that the file is open and must do permission checking on each read and write call. On a local file system, a user can open a file and then change the permissions so that no one is allowed to touch it, but will still be able to write to the file because it is open. On a remote file system, by contrast, the write would fail. To get around this problem, the server's permission checking algorithm should allow the owner of a file to access it regardless of the permission setting.

A similar problem has to do with paging in from a file over the network. The operating system usually checks for execute permission before opening a file for demand paging, and then reads blocks from the open file. The file may not have read permission, but after it is opened it doesn't matter. An NFS server can not tell the difference between a normal file read and a demand page-in read. To make this work, the server allows reading of files if the "uid" given in the call has execute or read permission on the file.

In most operating systems, a particular user (or the user ID zero) has access to all files no matter what permission and ownership they have. This "super-user" permission may not be allowed on the server, since anyone who can become super-user on their workstation could gain access to all remote files. The server by default maps user id 0 to -2 before doing its access checking. This works except for NFS root file systems, where super-user access cannot be avoided.

Setting RPC parameters

Various file system parameters and options should be set at mount time. The mount protocol is described in the section "Mount protocol definition" on page 340. For example, soft as well as hard mounts are usually both provided. Soft mounted file systems return errors when RPC operations fail (after a given number of optional retransmissions), while hard mounted file systems continue to retransmit forever. Clients and servers may need to keep caches of recent operations to help avoid problems with non-idempotent operations.

Mount protocol definition

The mount protocol is separate from, but related to, the NFS protocol. It provides operating system specific services to get the NFS started looking up server path names, validating user identities, and checking access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote file system.

The mount protocol is kept separate from the NFS protocol to make it easy to plug in new access checking and validation methods without changing the NFS server protocol.

Notice that the protocol definition implies stateful servers because the server maintains a list of client mount requests. The mount list information is not critical for the correct functioning of either the client or the server. It is intended for advisory use only, for example, to warn possible clients when a server is going down.

Version 1 of the mount protocol is used with version 2 of the NFS protocol. The only connecting point is the `fhandle` structure, which is the same for both protocols.

RPC information

Authentication:

The mount service uses AUTH_UNIX() and AUTH_DES() style authentication only.

Transport Protocols:

The mount service is currently supported on UDP/IP only.

Port Number:

Consult the server's portmapper, described in the chapter "Remote Procedure Calls: Protocol Specification" on page 345 to find the port number on which the mount service is registered.

Sizes of XDR structures

These are the sizes, given in decimal bytes, of various XDR structures used in the protocol:

```
/* The maximum number of bytes in a pathname
   argument */
const MNTPATHLEN = 1024;

/* The maximum number of bytes in a name
   argument */
const MNTNAMLEN = 255;

/* The size in bytes of the opaque file handle
 */
const FHSIZE = 32;
```

Basic data types

This section presents the data types used by the mount protocol. In many cases they are similar to the types used in NFS.

fhandle

```
typedef opaque fhandle[FHSIZE];
```

The type `fhandle` is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

This is the same as the fhandle XDR definition in version 2 of the NFS protocol; refer to the section "Basic data types" on page 323.

fhstatus

```
union fhstatus switch (unsigned status) {
   case 0:
      fhandle directory;
   default:
      void;
};
```

The type `fhstatus` is a union. If a status of zero is returned, the call completed successfully and a file handle for the directory follows. A non-zero status indicates some sort of error. In this case the status is an error number.

dirpath

```
typedef string dirpath<MNTPATHLEN>;
```

The type `dirpath` is a server pathname of a directory.

name

```
typedef string name<MNTNAMLEN>;
```

The type `name` is an arbitrary string used for various names.

Server procedures

The following sections define the RPC procedures supplied by a mount server.

```
/*
* Protocol description for the mount program
*/
program MOUNTPROG {
/*
* Version 1 of the mount protocol used with
* version 2 of the NFS protocol.
*/
   version MOUNTVERS {
      void  MOUNTPROC_NULL(void)  = 0;
      fhstatus  MOUNTPROC_MNT(dirpath) = 1;
      mountlist MOUNTPROC_DUMP(void)  = 2;
      void  MOUNTPROC_UMNT(dirpath) = 3;
      void  MOUNTPROC_UMNTALL(void) = 4;
      exportlist MOUNTPROC_EXPORT(void) = 5;
   } = 1;
} = 100005;
```

Do nothing

```
void
MNTPROC_NULL(void) = 0;
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

Add mount entry

```
fhstatus
MNTPROC_MNT(dirpath) = 1;
```

If the reply status is 0, then the reply directory contains the file handle for the directory dirname. This file handle may be used in the NFS protocol. This procedure also adds a new entry to the mount list for this client mounting dirname.

Return Mount Entries

```
struct *mountlist {
   name  hostname;
   dirpath directory;
   mountlist nextentry;
};

mountlist
MNTPROC_DUMP(void) = 2;
```

Returns the list of remote mounted file systems. `mountlist` contains one entry for each `hostname` and `directory` pair.

Remove mount entry

```
void
MNTPROC_UMNT(dirpath) = 3;
```

Removes the mount list entry for the input `dirpath`.

Remove all mount entries

```
void
MNTPROC_UMNTALL(void) = 4;
```

Removes all of the mount list entries for this client.

Return export list

```
struct *groups {
   name grname;
   groups grnext;
};

struct *exportlist {
   dirpath filesys;
   groups groups;
   exportlist next;
};

exportlist
MNTPROC_EXPORT(void) = 5;
```

Returns a variable number of export list entries. Each entry contains a file system name and a list of groups that are allowed to import it. The file system name is in `filesys`, and the group name is in the list groups.

Note

The `exportlist` should contain more information about the status of the file system, such as a read-only flag.

Remote Procedure Calls: Protocol Specification

19

Introduction

This chapter specifies a message protocol used in implementing Sun's Remote Procedure Call (RPC) package. The RPC protocol specification has been designated RFC 1050 by the ARPA-Internet Network Information Center.

The message protocol is specified with the External Data Representation (XDR) language. Refer to the chapter "XDR Standard: Protocol Specification" on page 371 for the details. Here, we assume that the reader is familiar with XDR and do not attempt to justify it or its uses). The paper by Birrell and Nelson [1] is recommended as an excellent background to and justification of RPC.

Terminology

This chapter discusses servers, services, programs, procedures, clients, and versions. These terms are defined as follows:

- *server*: a piece of software where network services are implemented
- *network service*: a collection of one or more remote programs
- *remote program*: implements one or more remote procedures. The procedures, their parameters, and results are documented in the specific program's protocol specification (see the Port Mapper Program Protocol, below, for an example).
- *network clients*: pieces of software that initiate remote procedure calls to services. A server may support more than one version of a remote program in order to be forward compatible with changing protocols.

For example, imagine two programs that compose a network file service. One program deals with high-level applications such as file system access control and locking. The other deals with low-level file IO and has procedures such as "read" and "write." A client machine of the network file service calls the procedures associated with the two programs of the service on behalf of a user on the client machine.

The RPC model

The remote procedure call model is similar to the local procedure call model. In the local procedure call model, the caller places arguments to a procedure in a specified location. The caller transfers control to the procedure, which it eventually regains. At that point, the caller extracts the results of the procedure from the specified location and continues execution.

The remote procedure call is similar, in that one thread of control logically winds through two processes—one is the caller's process, the other is a server's process. That is, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the caller extracts the results of the procedure, and resumes execution.

On the server side, a process is dormant while awaiting the arrival of a call message. When a call message arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and awaits the next call message.

Note that in this model, only one of the two processes is active at any given time. However, we give this model only as an example. The RPC protocol makes no restrictions on the implemented concurrency model, so others are possible. For example, an implementation may have asynchronous RPC calls, so that the client can do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request, so that the server can be free to receive other requests.

Transports and semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol deals only with specification and interpretation of messages.

It is important to point out that RPC does not try to implement any kind of reliability and that the application must be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP/IP[6], then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP/IP[7], it must implement is own retransmission and time-out policy as the RPC layer does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport such as UDP/IP. If an application retransmits RPC messages after short time-outs, the only thing it can infer if it receives no reply is that the procedure was executed zero or more times. If it does receive a reply, then it can infer that the procedure was executed at least once.

A server may wish to remember previously granted requests from a client and not regrant them in order to insure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request. The main use of this transaction is by the client RPC layer in matching replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, knowing this fact, can choose to remember this ID after granting a request and not regrant requests with the same ID in order to achieve some degree of execute-at-most-once semantics. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a reliable transport such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

There are other possibilities for transports besides datagram- or connection-oriented protocols. For example, a request-reply protocol such as VMTP[2] is perhaps the most natural transport for RPC.

Note

In ConvexOS, an application can use either TCP/IP or UDP/IP transport services.

Binding and rendezvous independence

The act of binding a client to a service is NOT part of the remote procedure call specification. This important and necessary function is left up to some higher-level software. (The software may use RPC itself—see "Port mapper program protocol" on page 366).

Implementors should think of the RPC protocol as the jump-subroutine instruction ("JSR") of a network; the loader (binder) makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the network makes RPC useful, using RPC to accomplish this task.

Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice-versa. Security and access control mechanisms can be built on top of the message authentication. Several different authentication protocols can be supported. A field in the RPC header indicates which protocol is being used. More information on specific authentication protocols can be found in section "Authentication Protocols" on page 356.

RPC protocol requirements

The RPC protocol must provide for the following:

- Unique specification of a procedure to be called
- Provisions for matching response messages to request messages
- Provisions for authenticating the caller to service and vice-versa

Besides these requirements, features that detect the following are worth supporting because of protocol roll-over errors, implementation bugs, user error, and network administration:

- RPC protocol mismatches
- Remote program protocol version mismatches
- Protocol errors (such as misspecification of a procedure's parameters)
- Reasons why remote authentication failed
- Any other reasons why the desired procedure was not called

Programs and procedures

The RPC call message has three unsigned fields: remote program number, remote program version number, and remote procedure number. The three fields uniquely identify the procedure to be called. Program numbers are administered by some central authority. Once an implementor has a program number, she can implement her remote program; the first implementation would most likely have the version number of 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification may state that its procedure number 5 is "read" and procedure number 12 is "write."

Just as remote program protocols can change over several versions, the actual RPC message protocol can also change. Therefore, the call message also has in it the RPC version number, which is always equal to 2 for the current version of RPC.

The reply message to a request message has enough information to distinguish the following error conditions:

- The remote implementation of RPC does speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist. (This is usually a caller side protocol or programming error.)
- The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is usually caused by a disagreement about the protocol between client and service.)

Authentication

Provisions for authentication of caller to service and vice-versa are provided as a part of the RPC protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
   AUTH_NULL         = 0,
   AUTH_UNIX         = 1,
   AUTH_SHORT        = 2,
   AUTH_DES          = 3
   /* and more to be defined */
};

struct opaque_auth {
   auth_flavor flavor;
   opaque body<400>;
};
```

Any `opaque_auth` structure is an `auth_flavor` enumeration followed by bytes which are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. (See section "Authentication Protocols" on page 356 for definitions of the various authentication protocols.)

If authentication parameters are rejected, the response message contains information stating why.

Program number assignment

Program numbers are given out in groups of 0x20000000 (decimal 536870912) according to Table 26:

Table 26 Program number assignments

Program numbers	Description
0 - 1fffffff	Defined by Sun
20000000 - 3fffffff	Defined by user
40000000 - 5fffffff	Transient
60000000 - 7fffffff	Reserved
80000000 - 9fffffff	Reserved
a0000000 - bfffffff	Reserved
c0000000 - dfffffff	Reserved
e0000000 - ffffffff	Reserved

The first group is a range of numbers administered by Sun Microsystems and should be identical for all sites. The second range is for applications peculiar to a particular site. This range is intended primarily for debugging new programs. When a site develops an application that might be of general interest, that application should be given an assigned number in the first range. The third group is for applications that dynamically generate program numbers. The final groups are reserved for future use, and should not be used.

Other uses of the RPC protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which other (non-RPC) protocols can be implemented. Sun currently uses the RPC message protocol for the following two (non-RPC) protocols: batching (or pipelining) and broadcast RPC. These two protocols are discussed but not defined below.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server. Batching typically uses reliable byte stream protocols (like TCP/IP) for its transport. In the case of batching, the client never waits for a reply from the server, and the server does not send replies to batch requests. A sequence of batch calls is usually terminated by a legitimate RPC in order to flush the pipeline (with positive acknowledgment).

Broadcast RPC

In broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet-based protocols (like UDP/IP) as its transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors. Broadcast RPC uses the Port Mapper RPC service to achieve its semantics. Refer to "Port mapper program protocol" on page 366 for more information.

The RPC message protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top-down style.

```
enum msg_type {
   CALL  = 0,
   REPLY = 1
};

/*
 * A reply to a call message can take on two forms: The message was
 * either accepted or rejected.
 */
enum reply_stat {
   MSG_ACCEPTED = 0,
   MSG_DENIED   = 1
};

/*
 * Given that a call message was accepted, the following is the
 * status of an attempt to call a remote procedure.
 */
enum accept_stat {
   SUCCESS       = 0,      /* RPC executed successfully */
   PROG_UNAVAIL  = 1,      /* remote hasn't exported program */
   PROG_MISMATCH = 2,      /* remote can't support version */
   PROC_UNAVAIL  = 3,      /* program can't support procedure */
   GARBAGE_ARGS  = 4       /* procedure can't decode params */
};

/*
 * Reasons why a call message was rejected:
 */
```

```
enum reject_stat {
   RPC_MISMATCH = 0,    /* RPC version number != 2          */
   AUTH_ERROR = 1       /* remote can't authenticate caller */
};

/*
 * Why authentication failed:
 */
enum auth_stat {
   AUTH_BADCRED      = 1,     /* bad credentials */
   AUTH_REJECTEDCRED = 2,     /* client must begin new session */
   AUTH_BADVERF      = 3,     /* bad verifier */
   AUTH_REJECTEDVERF = 4,     /* verifier expired or replayed  */
   AUTH_TOOWEAK      = 5      /* rejected for security reasons */
};

/*
 * The  RPC  message:
 * All   messages  start with   a transaction  identifier,  xid,
 * followed  by a  two-armed  discriminated union.   The union's
 * discriminant is a  msg_type which switches to  one of the two
 * types   of the message.   The xid  of a REPLY  message always
 * matches  that of the initiating CALL   message.   NB: The xid
 * field is only  used for clients  matching reply messages with
 * call messages  or for servers detecting  retransmissions;
 * the service side  cannot treat this id  as any type of
 * sequence number.
 */
struct rpc_msg {
   unsigned int xid;
   union switch (msg_type mtype) {
      case CALL:
         call_body cbody;
      case REPLY:
         reply_body rbody;
   } body;
};

/*
 * Body of an RPC request call:
 * In version 2 of the  RPC protocol specification, rpcvers must
 * be equal to 2.  The  fields prog,  vers, and proc specify the
 * remote program, its version number, and the  procedure with
 * in the remote program to be called.  After these  fields are
 * two authentication  parameters:
 *     cred (authentication credentials)
 *     verf  (authentication verifier).
 * The  two authentication parameters are followed by the
```

```
 * parameters to the remote procedure, which are specified
 * by the specific program protocol.
 */
struct call_body {
   unsigned int rpcvers;      /* must be equal to two (2) */
   unsigned int prog;
   unsigned int vers;
   unsigned int proc;
   opaque_auth cred;
   opaque_auth verf;
   /* procedure specific parameters start here */
};

/*
 * Body of a reply to an RPC request:
 * The call message was either accepted or rejected.
 */
union reply_body switch (reply_stat stat) {
   case MSG_ACCEPTED:
      accepted_reply areply;
   case MSG_DENIED:
      rejected_reply rreply;
} reply;

/*
 * Reply to an RPC request that was accepted by the server
 * there could be an error even though the request was
 * accepted.
 * The first field is an authentication verifier that the server
 * generates in order to validate itself to the caller. It is
 * followed by a union whose discriminant is an enum
 * accept_stat. The SUCCESS arm of the union is protocol
 * specific. The PROG_UNAVAIL, PROC_UNAVAIL, and GARBAGE_ARGP
 * arms of the union are void. The PROG_MISMATCH arm specifies
 * the lowest and highest version numbers of the remote program
 * supported by the server.
 */
struct accepted_reply {
   opaque_auth verf;
   union switch (accept_stat stat) {
      case SUCCESS:
      /* procedure-specific results start here */
         opaque results[0];
      case PROG_MISMATCH:
         struct {
            unsigned int low;
            unsigned int high;
         } mismatch_info;
```

```
      default:
         /* Void.  Cases include PROG_UNAVAIL, PROC_UNAVAIL, *
          *  and GARBAGE_ARGS.
          */
         void;
   } reply_data;
};

/*
 * Reply to an RPC request that was rejected by the server:
 * The request can be rejected for two reasons: either the
 * server is not running a compatible version of the RPC
 * protocol (RPC_MISMATCH), or the server refuses to
 * authenticate the caller (AUTH_ERROR). In case of an RPC
 * version mismatch, the server returns the lowest and highest
 * supported RPC version numbers. In case of refused
 * authentication, failure status is returned.
 */
union rejected_reply switch (reject_stat stat) {
   case RPC_MISMATCH:
      struct {
         unsigned int low;
         unsigned int high;
      } mismatch_info;
   case AUTH_ERROR:
      auth_stat stat;
};
```

Authentication Protocols

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some "flavors" of authentication implemented at (and supported by) Sun. Other sites are free to invent new authentication types, with the same rules of flavor number assignment as there is for program number assignment.

Null authentication

Often calls must be made where the caller does not know who he is or the server does not care who the caller is. In this case, the flavor value (the discriminant of the `opaque_auth`'s union) of the RPC message's credentials, verifier, and response verifier is `AUTH_NULL`. The bytes of the `opaque_auth`'s body are undefined. It is recommended that the opaque length be zero.

UNIX authentication

The caller of a remote procedure may wish to identify himself as he is identified on a UNIX system. The value of the credential's discriminant of an RPC call message is `AUTH_UNIX`. The bytes of the credential's opaque body encode the following structure:

```
struct auth_unix {"
   unsigned int stamp;
   string machinename<255>;
   unsigned int uid;
   unsigned int gid;
   unsigned int gids<10>;
};
```

The stamp is an arbitrary ID which the caller machine may generate. The machinename is the name of the caller's machine (like "krypton"). The uid is the caller's effective user ID. The gid is the caller's effective group ID. The gids is a counted array of groups which contain the caller as a member. The verifier accompanying the credentials should be of `AUTH_NULL` (defined above).

The value of the discriminant of the response verifier received in the reply message from the server may be `AUTH_NULL` or `AUTH_SHORT`. In the case of `AUTH_SHORT` the bytes of the response verifier's string encode an opaque structure. This new opaque structure may now be passed to the server instead of the

original AUTH_UNIX flavor credentials. The server keeps a cache which maps shorthand opaque structures (passed back by way of an AUTH_SHORT style response verifier) to the original credentials of the caller. The caller can save network bandwidth and server cpu cycles by using the new credentials.

The server may flush the shorthand opaque structure at any time. If this happens, the remote procedure call message will be rejected due to an authentication error. The reason for the failure will be AUTH_REJECTEDCRED. At this point, the caller may wish to try the original AUTH_UNIX style of credentials.

DES authentication

UNIX authentication suffers from two major problems:

- The naming is too UNIX-system oriented.
- There is no verifier, so credentials can easily be faked.

DES authentication attempts to fix these two problems.

Naming

The first problem is handled by addressing the caller by a simple string of characters instead of by an operating system specific integer. This string of characters is known as the "netname" or network name of the caller. The server is not allowed to interpret the contents of the caller's name in any other way except to identify the caller. Thus, netnames should be unique for every caller in the internet.

It is up to each operating system's implementation of DES authentication to generate netnames for its users that insure this uniqueness when they call upon remote servers. Operating systems already know how to distinguish users local to their systems. It is usually a simple matter to extend this mechanism to the network. For example, a user at Sun with a user ID of 515 might be assigned the following netname: "unix.515@sun.com." This netname contains three items that serve to insure it is unique. Going backwards, there is only one naming domain called "sun.com" in the Internet. Within this domain, there is only one user with user ID 515. However, there may be another user on another operating system, for example VMS, within the same naming domain that, by coincidence, happens to have the same user ID. To insure that these two users can be distinguished we add the operating system name. So one user is "unix.515@sun.com" and the other is "vms.515@sun.com".

The first field is actually a naming method rather than an operating system name. It just happens that today there is almost a one-to-one correspondence between naming methods and operating systems. If the world could agree on a naming standard, the first field could be the name of that standard, instead of an operating system name.

DES authentication verifiers

Unlike UNIX authentication, DES authentication does have a verifier so the server can validate the client's credential (and vice-versa). The contents of this verifier is primarily an encrypted timestamp. The server can decrypt this timestamp, and if it is close to what the real time is, then the client must have encrypted it correctly. The only way the client could encrypt it correctly is to know the "conversation key" of the RPC session. And if the client knows the conversation key, then it must be the real client.

The conversation key is a DES [5] key which the client generates and notifies the server of in its first RPC call. The conversation key is encrypted using a public key scheme in this first transaction. The particular public key scheme used in DES authentication is Diffie-Hellman [3] with 192-bit keys. The details of this encryption method are described later.

The client and the server need the same notion of the current time in order for all of this to work. If network time synchronization cannot be guaranteed, then client can synchronize with the server before beginning the conversation, perhaps by consulting the Internet Time Server (TIME[4]).

The way a server determines if a client timestamp is valid is somewhat complicated. For any other transaction but the first, the server just checks for two things:

- The timestamp is greater than the one previously seen from the same client.
- The timestamp has not expired.

A timestamp is expired if the server's time is later than the sum of the client's timestamp plus what is known as the client's "window." The "window" is a number the client passes (encrypted) to the server in its first transaction. You can think of it as a lifetime for the credential.

This explains everything but the first transaction. In the first transaction, the server checks only that the timestamp has not expired. If this was all that was done though, then it would be quite easy for the client to send random data in place of the timestamp with a fairly good chance of succeeding. As an added check, the client sends an encrypted item in the first transaction known as the "window verifier" which must be equal to the window minus 1, or the server will reject the credential.

The client too must check the verifier returned from the server to be sure it is legitimate. The server sends back to the client the encrypted timestamp it received from the client, minus one second. If the client gets anything different than this, it will reject it.

Nicknames and clock synchronization

After the first transaction, the server's DES authentication subsystem returns in its verifier to the client an integer "nickname" which the client may use in its further transactions instead of passing its netname, encrypted DES key and window every time. The nickname is most likely an index into a table on the server which stores for each client its netname, decrypted DES key and window.

Although they originally were synchronized, the client's and server's clocks can get out of synchronization again. When this happens the client RPC subsystem most likely will get back `RPC_AUTHERROR` at which point it should resynchronize.

A client may still get the `RPC_AUTHERROR` error even though it is synchronized with the server. The reason is that the server's nickname table is a limited size, and it may flush entries whenever it wants. A client should resend its original credential in this case and the server will give it a new nickname. If a server crashes, the entire nickname table gets flushed, and all clients will have to resend their original credentials.

```
/*
 * There are two kinds of credentials: one in which the client
 * uses its full network name, and one in which it uses its
 * "nickname"(unsigned integer) given to it by the server. The
 * client must use its full name in its first transaction with
 * the server, in which the server will return to the client its
 * nickname. The client may use its nickname in all further
 * transactions with the server. There is no requirement to use
 * the nickname, but it is wise to use it for performance
 * reasons.
 */
enum authdes_namekind {
   ADN_FULLNAME = 0,
   ADN_NICKNAME = 1
};

/* A 64-bit block of encrypted DES data */
typedef opaque des_block[8];

/* Maximum length of a network user's name */
const MAXNETNAMELEN = 255;

/*
 * A fullname contains the network name of the client, an
 * encrypted conversation key and the window.  The window is
 * actually a lifetime for the credential.  If the time
 * indicated in the verifier timestamp plus the window has past,
 * then the server should expire the request and not grant it.
 * To insure that requests are not replayed, the server should
 * insist that timestamps are greater than the previous one
 * seen, unless it is the first transaction.  In the first
 * transaction, the server checks instead that the window
 * verifier is one less than the window.
 */
struct authdes_fullname {
   string name<MAXNETNAMELEN>;   /* name of client */
   des_block key;                /* PK encrypted conversation key */
   unsigned int window;          /* encrypted window */
};

/*
 * A credential is either a fullname or a nickname
 */
union authdes_cred switch (authdes_namekind adc_namekind) {
   case ADN_FULLNAME:
      authdes_fullname adc_fullname;
   case ADN_NICKNAME:
```

```
        unsigned int adc_nickname;
};

/*
 * A timestamp encodes the time since midnight,
 * January 1, 1970.
 */
struct timestamp {
    unsigned int seconds;       /* seconds */
    unsigned int useconds;      /* and microseconds */ ;
};

/*
 * Verifier: client variety
 * The window verifier is only used in the first transaction.  In
 * conjunction with a fullname credential, these items are
 * packed into the following structure before being encrypted:
 *
 * struct {
 *      adv_timestamp;              -- one DES block
 *      adc_fullname.window;        -- one half DES block
 *      adv_winverf;                -- one half DES block
 * }
 * This structure is encrypted using CBC mode encryption with
 * an input vector of zero.  All other encryptions of
 * timestamps use ECB mode encryption.
 */
 struct authdes_verf_clnt {
    timestamp adv_timestamp;    /* encrypted timestamp       */
    unsigned int adv_winverf;   /* encrypted window verifier */
};

/*
 * Verifier: server variety
 * The server returns (encrypted) the same timestamp the client
 * gave it minus one second.  It also tells the client its
 * nickname to be used in future transactions (unencrypted).
 */
struct authdes_verf_svr {
    timestamp adv_timeverf;     /* encrypted verifier      */
    unsigned int adv_nickname; /* new nickname for client */
};
```

Diffie-Hellman encryption

In this scheme, there are two constants, BASE and MODULUS. The particular values Sun has chosen for these for the DES authentication protocol are:

```
const BASE = 3;
const MODULUS=
"d4a0ba0250b6fd2ec626e7efd6df76c716e22d0944b88b
"; /* hex */
```

The way this scheme works is best explained by an example. Suppose there are two people "A" and "B" who want to send encrypted messages to each other. So, A and B both generate "secret" keys at random which they do not reveal to anyone. Let these keys be represented as SK(A) and SK(B). They also publish in a public directory their "public" keys. These keys are computed as follows:

```
PK(A) = ( BASE ** SK(A) ) mod MODULUS
PK(B) = ( BASE ** SK(B) ) mod MODULUS
```

The "**" notation is used here to represent exponentiation. Now, both A and B can arrive at the "common" key between them, represented here as CK(A, B), without revealing their secret keys.

A computes:

```
CK(A, B) = (PK(B) ** SK(A)) mod MODULUS
```

while B computes:

```
CK(A, B) = (PK(A) ** SK(B)) mod MODULUS
```

These two can be shown to be equivalent:

```
PK(B) ** SK(A)) mod MODULUS =
(PK(A) **SK(B)) mod MODULUS
```

We drop the "mod MODULUS" parts and assume modulo arithmetic to simplify things:

```
PK(B) ** SK(A) = PK(A) ** SK(B)
```

Then, replace PK(B) by what B computed earlier and likewise for PK(A).

```
(BASE ** SK(B)) ** SK(A)) = ((BASE ** SK(A))
** SK(B))
```

which leads to:

```
BASE ** (SK(A) * SK(B)) = BASE ** (SK(A) *
SK(B))
```

This common key CK(A, B) is not used to encrypt the domesticates used in the protocol. Rather, it is used only to encrypt a conversation key which is then used to encrypt the timestamps. The reason for doing this is to use the common key as little as possible, for fear that it could be broken. Breaking the conversation key is a far less serious offense, since conversations are relatively short-lived.

The conversation key is encrypted using 56-bit DES keys, yet the common key is 192 bits. To reduce the number of bits, 56 bits are selected from the common key as follows. The middle-most 8-bytes are selected from the common key, and then parity is added to the lower order bit of each byte, producing a 56-bit key with 8 bits of parity.

Record marking standard

When RPC messages are passed on top of a byte stream protocol (like TCP/IP), it is necessary, or at least desirable, to delimit one message from another in order to detect and possibly recover from user protocol errors. This is called record marking (RM). Sun uses this RM/TCP/IP transport for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by 0 to (2**31) - 1 bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values—a boolean which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the highest-order bit of the header; the length is the 31 low-order bits. (Note that this record specification is NOT in XDR standard form!)

The RPC language

Just as there was a need to describe the XDR data-types in a formal language, there is also need to describe the procedures that operate on these XDR data-types in a formal language as well. We use the RPC Language for this purpose. It is an extension to the XDR language. The following example is used to describe the essence of the language.

An example service described in the RPC language

Here is an example of the specification of a simple ping program.

```
/*
 * Simple ping program
 */
program PING_PROG {
   /* Latest and greatest version */
   version PING_VERS_PINGBACK {
   void
   PINGPROC_NULL(void) = 0;
   /*
   * Ping the caller, return the round-trip time
   * (in microseconds). Returns -1 if the
   * operation timed out.
   */
   int
   PINGPROC_PINGBACK(void) = 1;
} = 2;

/*
 * Original version
 */
version PING_VERS_ORIG {
   void
   PINGPROC_NULL(void) = 0;
   } = 1;
} = 1;

const PING_VERS = 2;        /* latest version */
```

The first version described is `PING_VERS_PINGBACK()` with two procedures, `PINGPROC_NULL()` and `PINGPROC_PINGBACK()`. `PINGPROC_NULL()` takes no arguments and returns no results, but it is useful for computing round-trip times from the client to the server and back again. By convention, procedure 0 of any RPC protocol should have the

same semantics, and never require any kind of authentication. The second procedure is used for the client to have the server do a reverse ping operation back to the client, and it returns the amount of time (in microseconds) that the operation used. The next version, `PING_VERS_ORIG()` is the original version of the protocol and it does not contain `PINGPROC_PINGBACK()` procedure. It is useful for compatibility with old client programs, and as this program matures it may be dropped from the protocol entirely.

The RPC language specification

The RPC language is identical to the XDR language, except for the added definition of a program-def described below.

```
program-def:
   "program" identifier "{"
      version-def
      version-def *
   "}" "=" constant ";"

version-def:
   "version" identifier "{"
      procedure-def
      procedure-def *
   "}" "=" constant ";"

procedure-def:
   type-specifier identifier
      "(" type-specifier ")"
   "=" constant ";"
```

Syntax notes

- The following keywords are added and cannot be used as identifiers: "program" and "version."
- A version name cannot occur more than once within the scope of a program definition. Nor can a version number occur more than once within the scope of a program definition.
- A procedure name cannot occur more than once within the scope of a version definition. Nor can a procedure number occur more than once within the scope of version definition.

- Program identifiers are in the same name space as constant and type identifiers.
- Only unsigned constants can be assigned to programs, versions and procedures.

Port mapper program protocol

The port mapper program maps RPC program and version numbers to transport-specific port numbers. This program makes dynamic binding of remote programs possible.

This is desirable because the range of reserved port numbers is very small and the number of potential remote programs is very large. By running only the port mapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the port mapper.

The port mapper also aids in broadcast RPC. A given RPC program will usually have different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The port mapper, however, does have a fixed port number. So, to broadcast to a given program, the client actually sends its message to the port mapper located at the broadcast address. Each port mapper that picks up the broadcast then calls the local service specified by the client. When the port mapper gets the reply from the local service, it sends the reply on back to the client.

Port mapper protocol specification (in RPC language)

```
const PMAP_PORT = 111;          /* portmapper
                                 * port number */

/*
* A mapping of (program, version, protocol)
* to port number
*/
struct mapping {
   unsigned int prog;
   unsigned int vers;
   unsigned int prot;
   unsigned int port;
};

/*
 * Supported values for the "prot" field
```

```
 */
const IPPROTO_TCP = 6;  /* protocol number for
                         * TCP/IP */
const IPPROTO_UDP = 17; /* protocol number for
                         * UDP/IP */

/*
 * A list of mappings
 */
struct *pmaplist {
   mapping map;
   pmaplist next;
};

/*
 * Arguments to callit
 */
struct call_args {
   unsigned int prog;
   unsigned int vers;
   unsigned int proc;
   opaque args<>;
};

/*
 * Results of callit
 */
struct call_result {
   unsigned int port;
   opaque res<>;
};

/*
 * Port mapper procedures
 */
program PMAP_PROG {
   version PMAP_VERS {
      void PMAPPROC_NULL(void) = 0;
      bool PMAPPROC_SET(mapping) = 1;
      bool PMAPPROC_UNSET(mapping) = 2;
      unsigned int PMAPPROC_GETPORT(mapping) = 3;
      pmaplist PMAPPROC_DUMP(void) = 4;
      call_result PMAPPROC_CALLIT(call_args) = 5;
   } = 2;
} = 100000;
```

Port mapper operation

The port mapper program currently supports two protocols (UDP/IP and TCP/IP). The port mapper is contacted by talking to it on assigned port number 111 (SUNRPC [8]) on either of these protocols. The following is a description of each of the port mapper procedures:

`PMAPPROC_NULL:`

This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results.

`PMAPPROC_SET:`

When a program first becomes available on a machine, it registers itself with the port mapper program on the same machine. The program passes its program number "prog," version number "vers," transport protocol number "prot," and the port "port" on which it awaits service request. The procedure returns a boolean response whose value is TRUE if the procedure successfully established the mapping and FALSE otherwise. The procedure refuses to establish a mapping if one already exists for the tuple "(prog, vers, prot)".

`PMAPPROC_UNSET:`

When a program becomes unavailable, it should unregister itself with the port mapper program on the same machine. The parameters and results have meanings identical to those of PMAPPROC_SET. The protocol and port number fields of the argument are ignored.

`PMAPPROC_GETPORT:`

Given a program number "prog," version number "vers," and transport protocol number "prot," this procedure returns the port number on which the program is awaiting call requests. A port value of zeros means the program has not been registered. The "port" field of the argument is ignored.

`PMAPPROC_DUMP:`

This procedure enumerates all entries in the port mapper's database. The procedure takes no parameters and returns a list of program, version, protocol, and port values.

`PMAPPROC_CALLIT:`

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. It is intended for supporting broadcasts to arbitrary remote programs via the well-known port mapper's port. The parameters `prog`, `vers`, `proc`, and the bytes of `args` are the program number, version number, procedure number, and parameters of the remote procedure.

Notes

1. This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise.

2. The port mapper communicates with the remote program using UDP/IP only.

The procedure returns the remote program's port number, and the bytes of results are the results of the remote procedure.

References

[1] Birrell, Andrew D. & Nelson, Bruce Jay; "Implementing Remote Procedure Calls;" XEROX CSL-83-7, October 1983.

[2] Cheriton, D.; "VMTP: Versatile Message Transaction Protocol," Preliminary Version 0.3; Stanford University, January 1987.

[3] Diffie & Hellman; "New Directions in Cryptography;" IEEE Transactions on Information Theory IT-22, November 1976.

[4] Harrenstien, K.; "Time Server," RFC 738; Information Sciences Institute, October 1977.

[5] National Bureau of Standards; "Data Encryption Standard;" Federal Information Processing Standards Publication 46, January 1977.

[6] Postel, J.; "Transmission Control Protocol - DARPA Internet Program Protocol Specification," RFC 793; Information Sciences Institute, September 1981.

[7] Postel, J.; "User Datagram Protocol," RFC 768; Information Sciences Institute, August 1980.

[8] Reynolds, J. & Postel, J.; "Assigned Numbers," RFC 923; Information Sciences Institute, October 1984.

XDR Standard: Protocol Specification

20

This chapter contains the XDR protocol specification, which has been designated RFC1014 by the ARPA Network Information Center.

Introduction

The External Data Representation (XDR) standard is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the Sun Workstation, VAX, IBM-PC, and CONVEX machines. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used only to describe data. It is not a programming language. This language allows you to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. The XDR language itself is similar to the C language [1], just as Courier [4] is similar to Mesa. Protocols such as Sun RPC (Remote Procedure Call) and the NFS (Network File System) use XDR to describe the format of their data.

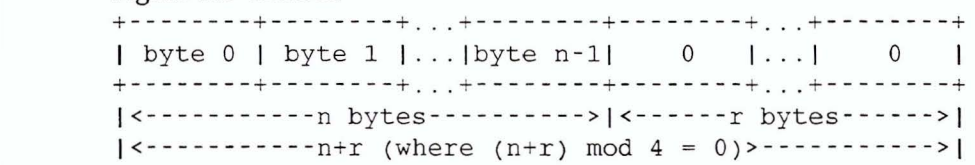
The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning. For example, the Ethernet standard suggests that bytes be encoded in "little-endian" style [2], or least significant bit first.

Basic block size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through n-1. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. If the n bytes needed to contain the data are not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of 4.

We include the familiar graphic box notation for illustration and comparison. In most illustrations, each box (delimited by a plus sign at the 4 corners and vertical bars and dashes) depicts a byte. Ellipses (...) between boxes show zero or more additional bytes where required.

Figure 100 A block



XDR data types

Each section that follows describes a data type defined in the XDR standard. Each section shows how the data type is declared in the language, and includes a graphic illustration of its encoding.

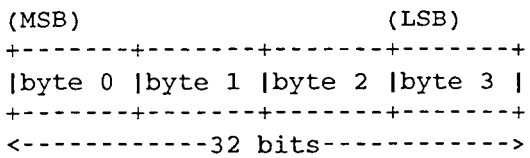
For each data type in the language we show a general paradigm declaration. Note that angle brackets (<>) denote variable length sequences of data and square brackets ([ ]) denote fixed-length sequences of data. "n," "m," and "r" denote integers. For the full language specification and more formal definitions of terms such as "identifier" and "declaration," refer to section "The XDR language specification" on page 386.

For some data types, more specific examples are included. A more extensive example of a data description is in section "Example of an XDR data description" on page 390.

Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648,2147483647]. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. Integers are declared as follows:

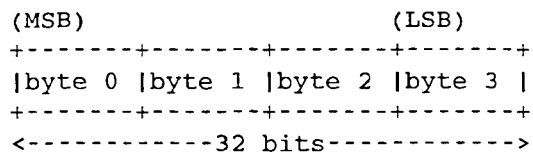
Figure 101 Integer



Unsigned integer

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0,4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. An unsigned integer is declared as follows:

Figure 102 Unsigned integer



Enumeration

Enumerations have the same representation as signed integers. Enumerations are handy for describing subsets of the integers. Enumerated data is declared as follows:

```
enum { name-identifier = constant, ... }
   identifier;
```

For example, the three colors red, yellow, and blue could be described by an enumerated type:

```
enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

It is an error to encode as an enum any other integer than those that have been given assignments in the enum declaration.

Boolean

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are declared as follows:

```
bool identifier;
```

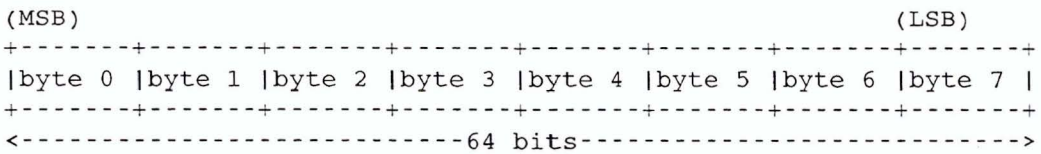
This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

Hyper integer and unsigned hyper integer

The standard also defines 64-bit (8-byte) numbers called hyper integer and unsigned hyper integer. Their representations are the obvious extensions of integer and unsigned integer defined above. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively. Their declarations:

Figure 103 Hyper integer and unsigned hyper integer



Floating-point

The standard defines the floating-point data type "float" (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers [3].

The floating-point number is described by:

$$(-1)^{S} \times 2^{E-Bias} \times 1.F$$

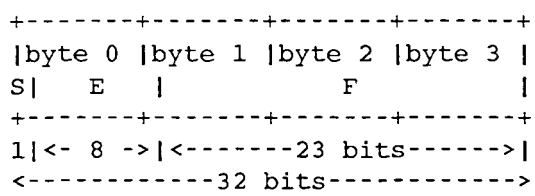
where the following three fields describe the single-precision floating-point number:

- S—The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.
- E—The exponent of the number, base 2. 8 bits are devoted to this field. The exponent is biased by 127.

- F—The fractional part of the number's mantissa, base 2. 23 bits are devoted to this field.

It is declared as follows:

Figure 104 Single-precision floating-point



Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating-point number are 0 and 31. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 9, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow) [3]. According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

Double-precision floating-point

The standard defines the encoding for the double-precision floating- point data type "double" (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized double-precision floating-point numbers [3]. The standard encodes the following three fields, which describe the double-precision floating-point number:

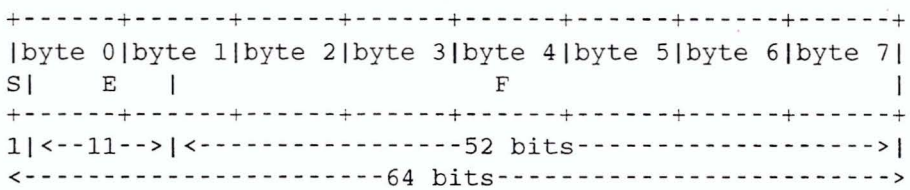
- S—The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.
- E—The exponent of the number, base 2. 11 bits are devoted to this field. The exponent is biased by 1023.
- F—The fractional part of the number's mantissa, base 2. 52 bits are devoted to this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as follows:

Figure 105 Double precision floating-point



Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a double-precision floating-point number are 0 and 63. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 12, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow) [3]. According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

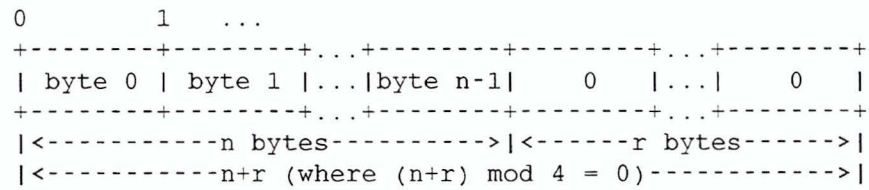
Fixed-length opaque data

At times, fixed-length uninterpreted data needs to be passed among machines. This data is called "opaque" and is declared as follows:

```
opaque identifier[n];
```

where the constant n is the (static) number of bytes necessary to contain the opaque data. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes r to make the total byte count of the opaque object a multiple of four.

Figure 106 Fixed-length opaque



Variable-length opaque data

The standard also provides for variable-length (counted) opaque data, defined as a sequence of n (numbered 0 through n-1) arbitrary bytes to be the number n encoded as an unsigned integer (as described below), and followed by the n bytes of the sequence.

Byte m of the sequence always precedes byte m+1 of the sequence, and byte 0 of the sequence always follows the sequence's length (count). enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Variable-length opaque data is declared in the following way:

```
opaque identifier<m>;
```

or

```
opaque identifier<>;
```

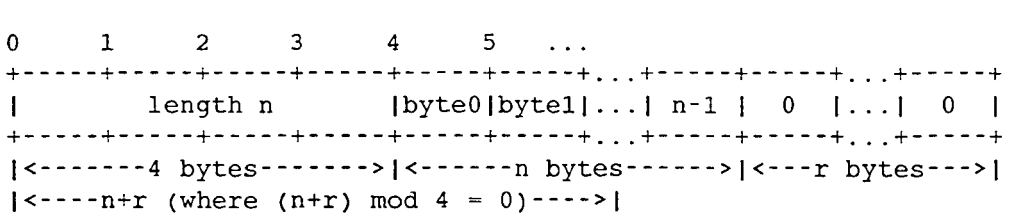
The constant m denotes an upper bound of the number of bytes that the sequence may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) - 1, the maximum length. The constant m would normally be found in a protocol specification.

For example, a filing protocol may state that the maximum data transfer size is 8192 bytes, as follows:

```
opaque filedata<8192>;
```

This can be illustrated as follows:

Figure 107 Variable-length opaque



It is an error to encode a length greater than the maximum described in the specification.

String

The standard defines a string of n (numbered 0 through n-1) ASCII bytes to be the number n encoded as an unsigned integer (as described above) followed by the n bytes of the string. Byte m of the string always precedes byte m+1 of the string, and byte 0 of the string always follows the string's length. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Counted byte strings are declared as follows:

```
string object<m>;
```

or

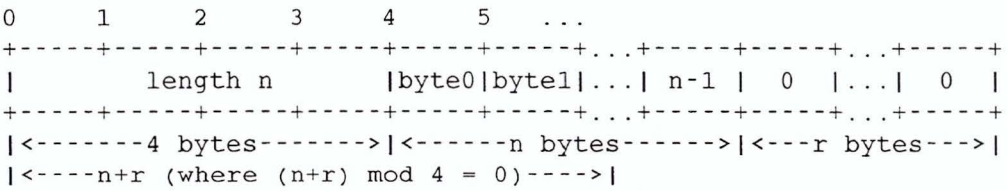
```
string object<>;
```

The constant m denotes an upper bound of the number of bytes that a string may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) - 1, the maximum length. The constant m is normally found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, as follows:

```
string filename<255>;
```

Which can be illustrated as:

Figure 108 A string



It is an error to encode a length greater than the maximum described in the specification.

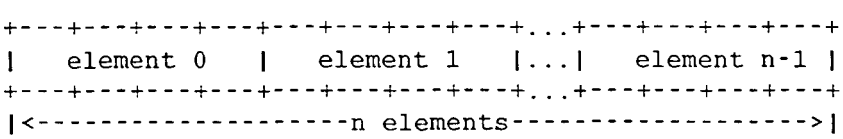
Fixed-length array

Declarations for fixed-length arrays of homogeneous elements are in the following form:

```
type-name identifier[n];
```

Fixed-length arrays of elements numbered 0 through n-1 are encoded by individually encoding the elements of the array in their natural order, 0 through n-1. Each element's size is a multiple of four bytes. Even though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type string, yet each element will vary in its length.

Figure 109 Fixed-length array



Variable-length array

Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count n (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element n- 1. The declaration for variable-length arrays follows the form

```
type-name identifier<m>;
```

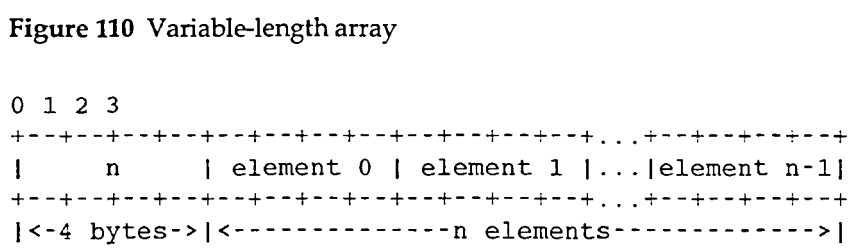
or

```
type-name identifier<>;
```

The constant m specifies the maximum acceptable element count of an array. If m is not specified, as in the second declaration, it is assumed to be

```
(2**32) - 1
```

Figure 110 Variable-length array



It is an error to encode a value of n that is greater than the maximum described in the specification.

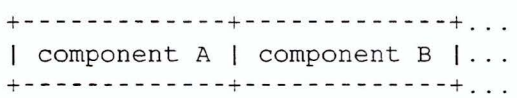
Structure

Structures are declared as follows:

```
struct {
    component-declaration-A;
    component-declaration-B;
        ...
} identifier;
```

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, even though the components may be different sizes.

Figure 111 A structure



Discriminated union

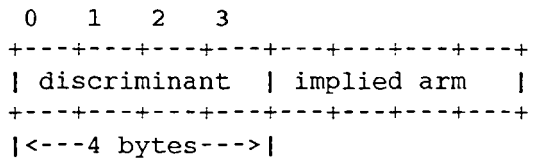
A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either int, unsigned int, or an enumerated type, such as bool. The component types are called arms of the union, and are preceded by the value of the discriminant which implies their encoding. Discriminated unions are declared as follows:

```
union switch (discriminant-declaration) {
    case discriminant-value-A:
    arm-declaration-A;
    case discriminant-value-B:
    arm-declaration-B;
    ...
    default: default-declaration;
} identifier;
```

Each case keyword is followed by a legal value of the discriminant. The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes.

The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

Figure 112 Discriminated union



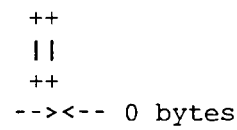
Void

An XDR void is a 0-byte quantity. Voids are useful for describing operations that don't take data as input or don't return data as output. They are also useful in unions, where some arms may contain data and others do not. The declaration is simply as follows:

```
void;
```

Voids are illustrated as follows:

Figure 113 Void



Constant

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

Const is used to define a symbolic name for a constant. It does not declare any data. The symbolic constant can be used anywhere a regular constant is used. For example, the following defines a symbolic constant DOZEN, equal to 12.

```
const DOZEN = 12;
```

Typedef

Typedef does not declare any data, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the typedef. For example, the following defines a new type called eggbox using an existing type called egg:

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name in the typedef if it is a variable. For example, the following two declarations are equivalent in declaring the variable fresheggs:

```
eggbox    fresheggs;
egg       fresheggs[DOZEN];
```

When a typedef involves a struct, enum, or union definition, there is another (preferred) syntax that may be used to define the same type. In general, a typedef of the following form:

```
typedef <<struct, union, or enum
   definition>> identifier;
```

can be converted to the alternative form by removing the typedef part and placing the identifier after the struct, union, or enum keyword, instead of at the end. For example, here are the two ways to define the type bool:

```
typedef enum {     /* using typedef */
   FALSE = 0,
   TRUE = 1
   } bool;

enum bool {        /* preferred alternative */
   FALSE = 0,
   TRUE = 1
   };
```

The reason this syntax is preferred is one does not have to wait until the end of a declaration to figure out the name of the new type.

Optional-data

Optional-data is one kind of union that occurs so frequently that we give it a special syntax of its own for declaring it. It is declared as follows:

```
type-name *identifier;
```

This is equivalent to the following union:

```
union switch (bool opted) {
   case TRUE:
   type-name element;
   case FALSE:
   void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, since the boolean opted can be interpreted as the length of the array:

```
type-name identifier<1>;
```

Optional-data is not so interesting in itself, but it is very useful for describing recursive data-structures such as linked-lists and trees. For example, the following defines a type stringlist that encodes lists of arbitrary length strings:

```
struct *stringlist {
   string item<>;
   stringlist next;
};
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
   case TRUE:
      struct {
         string item<>;
         stringlist next;
      } element;
   case FALSE:
      void;
};
```

or as a variable-length array:

```
struct stringlist<1> {
   string item<>;
   stringlist next;
};
```

Both of these declarations obscure the intention of the stringlist type, so the optional-data declaration is preferred over both of them. The optional-data type also has a close correlation to how recursive data structures are represented in high-level languages such as Pascal or C by use of pointers. In fact, the syntax is the same as that of the C language for pointers.

Areas for future enhancements

The XDR standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. Also missing are packed (or binary-coded) decimals.

The intent of the XDR standard was not to describe every kind of data that people have ever sent or will ever want to send from machine to machine. Rather, it only describes the most commonly used data-types of high-level languages such as Pascal or C, so that applications written in these languages will be able to communicate easily over some medium.

Perhaps you can imagine extensions to XDR that would let it describe almost any existing protocol, such as TCP. The minimum necessary for this is support for different block sizes and byte-orders. The XDR discussed here can then be considered the 4-byte big-endian member of a larger XDR family.

Discussion

Why a language for describing data?

There are many advantages in using a data-description language such as XDR, versus using diagrams. Languages are more formal than diagrams and lead to less ambiguous descriptions of data. Languages are also easier to understand, and allow you to think of other issues instead of the low-level details of bit-encoding. Also, there is a close analogy between the types of XDR and a high-level language such as C or Pascal. This makes the implementation of XDR encoding and decoding modules an easier task. Finally, the language specification itself is an ASCII string that can be passed from machine to machine to perform on-the-fly data interpretation.

Why only one byte-order for an XDR unit?

Supporting two byte-orderings requires a higher level protocol for determining in which byte-order the data is encoded. Since XDR is not a protocol, this can't be done. The advantage of this, however, is that data in XDR format can be written to a magnetic tape, for example, and any machine will be able to interpret it since no higher level protocol is necessary for determining the byte-order.

Why does XDR use big-endian byte-order?

Yes, it is unfair, but having only one byte-order means you have to be unfair to somebody. Many architectures, such as the Motorola 68000 and IBM 370, support the big-endian byte-order.

Why is the XDR unit four bytes wide?

There is a trade-off in choosing the XDR unit size. Choosing a small size such as two makes the encoded data small, but causes alignment problems for machines that aren't aligned on these boundaries. A large size such as eight means the data will be aligned on virtually every machine, but causes the encoded data to grow too big. We chose four as a compromise. Four is big enough to support most architectures efficiently, except for rare machines such as the eight-byte aligned Cray. Four is also small enough to keep the encoded data restricted to a reasonable size.

Why must variable-length data be padded with zeros?

It is desirable that the same data encode into the same thing on all machines, so that encoded data can be meaningfully compared or checksummed. Forcing the padded bytes to be zero ensures this.

Why is there no explicit data-typing?

Data-typing has a relatively high cost for what small advantages it may have. One cost is the expansion of data due to the inserted type fields. Another is the added cost of interpreting these type fields and acting accordingly. And most protocols already know what type they expect, so data-typing supplies only redundant information. However, one can still get the benefits of data-typing using XDR. One way is to encode two things: first a string which is the XDR data description of the encoded data, and then the encoded data itself. Another way is to assign a value to all the types in XDR, and then define a universal type which takes this value as its discriminant and for each value, describes the corresponding data type.

The XDR language specification

Notational conventions

This specification uses an extended Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

- The characters `| () [] " ,` and `*` are special.
- Terminal symbols are strings of any characters surrounded by double quotes.
- Non-terminal symbols are strings of non-special characters.
- Alternative items are separated by a vertical bar (`|`).
- Optional items are enclosed in brackets.
- Items are grouped together by enclosing them in parentheses.
- A `*` following an item means 0 or more occurrences of that item.

For example, consider the following pattern:

```
"a " "very" (", " " very")* [" cold " "and"]  "
rainy " ("day" | "night")
```

An infinite number of strings match this pattern. A few of them are:

a very rainy day

a very, very rainy day

a very cold and rainy day

a very, very, very cold and rainy night

Lexical notes

- Comments begin with '`/*`' and terminate with '`*/`', as shown below

```
/* This is a comment */
```

- White space serves to separate items and is otherwise ignored.

- An identifier is a letter followed by an optional sequence of letters, digits or underbars (_). The case of identifiers is not ignored.
- A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign (-).

Syntax information

```
declaration:
   type-specifier identifier
   | type-specifier identifier "[" value "]"
   | type-specifier identifier "<" [ value ] ">"
   | "opaque" identifier "[" value "]"
   | "opaque" identifier "<" [ value ] ">"
   | "string" identifier "<" [ value ] ">"
   | type-specifier "*" identifier
   | "void"

value:
   constant
   | identifier

type-specifier:
   [ "unsigned" ] "int"
   | [ "unsigned" ] "hyper"
   | "float"
   | "double"
   | "bool"
   | enum-type-spec
   | struct-type-spec
   | union-type-spec
   | identifier

enum-type-spec:
   "enum" enum-body

enum-body:
   "{"
   ( identifier "=" value )
   ( "," identifier "=" value )*
   "}"

struct-type-spec:
   "struct" struct-body
```

```
struct-body:
   "{"
   ( declaration ";" )
   ( declaration ";" )*
   "}"

union-type-spec:
   "union" union-body

union-body:
   "switch" "(" declaration ")" "{"
   ( "case" value ":" declaration ";" )
   ( "case" value ":" declaration ";" )*
   [ "default" ":" declaration ";" ]
   "}"

constant-def:
   "const" identifier "=" constant ";"

type-def:
   "typedef" declaration ";"
   | "enum" identifier enum-body ";"
   | "struct" identifier struct-body ";"
   | "union" identifier union-body ";"

definition:
   type-def
   | constant-def

specification:
   definition *
```

Syntax notes

- The following are keywords and cannot be used as identifiers:
  - bool
  - case
  - const
  - default
  - double
  - enum
  - float
  - hyper
  - opaque
  - string

- struct
- switch
- typedef
- union
- unsigned
- void

- Only unsigned constants can be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a const definition.
- Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.
- Similarly, variable names must be unique within the scope of struct and union declarations. Nested structure and union declarations create new scopes.
- The discriminant of a union must be of a type that evaluates to an integer. That is, int, unsigned int, bool, an enumerated type or any typedef type that evaluates to one of these is legal. Also, the case value must be one of the legal values of the discriminant. Finally, a case value may not be specified more than once within the scope of a union declaration.

Example of an XDR data description

Figure 114 shows a short XDR data description of a data structure named file, which might be used to transfer files from one machine to another.

Figure 114 XDR data description

```
const MAXUSERNAME = 32;     /* max length of a user name */
const MAXFILELEN = 65535;   /* max length of a file      */
const MAXNAMELEN = 255;     /* max length of a file name */

/*
 * Types of files:
 */
enum filekind {
    TEXT = 0,       /* ascii data */
    DATA = 1,       /* raw data   */
    EXEC = 2        /* executable */
};

/*
 * File information, per kind of file:
 */
union filetype switch (filekind kind) {
    case TEXT:
         void; /* no extra information */
    case DATA:
       string creator<MAXNAMELEN>; /* data creator */
    case EXEC:
       string interpretor<MAXNAMELEN>;/*program interpretor*/
};

/*
 * A complete file:
 */
struct file {
    string filename<MAXNAMELEN>; /* name of file */
    filetype type;               /* info about file */
    string owner<MAXUSERNAME>;   /* owner of file   */
    opaque data<MAXFILELEN>;     /* file data       */
};
```

Suppose now that there is a user named john who wants to store his lisp program, sillyprog, which only contains the statement

```
(quit)
```

His file would be encoded as shown in Table 27:

Table 27 Encoding example

Offset	Hex Bytes	ASCII	Description
0	00 00 00 09	....	Length of filename = 9
4	73 69 6c 6c	sill	Filename characters
8	79 70 72 6f	ypro	...and more characters ...
12	67 00 00 00	g...	... and 3 zero-bytes of fill
16	00 00 00 02	....	Filekind is EXEC = 2
20	00 00 00 04	....	Length of interpretor = 4
24	6c 69 73 70	lisp	Interpretor characters
28	00 00 00 04	....	Length of owner = 4
32	6a 6f 68 6e	john	Owner characters
36	00 00 00 06	....	Length of file data = 6
40	28 71 75 69	(qui	File data bytes ...
44	74 29 00 00	t)..	... and 2 zero-bytes of fill

References

[1] Brian W. Kernighan & Dennis M. Ritchie, "The C Programming Language," Bell Laboratories, Murray Hill, New Jersey, 1978.

[2] Danny Cohen, "On Holy Wars and a Plea for Peace," IEEE Computer, October 1981.

[3] "IEEE Standard for Binary Floating-Point Arithmetic," ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

[4] "Courier: The Remote Procedure Call Protocol," XEROX Corporation, XSIS 038112, December 1981.

NIS: Protocol Specification 21

Introduction and terminology

The Network Information System (NIS) is a network lookup service providing read access to a replicated database. The lookup service is provided by a set of NIS database servers. The client interface to this service uses the Remote Procedure Call (RPC) mechanism.

Translating or mapping a name to its value is one of the most common operations performed in computer systems. Common examples are translations of variable names to virtual memory addresses, the translation of user names to system IDs or list of capabilities, and the translation of network node names to Internet addresses. There are two fundamental read-only operations that can be performed on a map: match and enumerate. *Match* is to look up a name, called a *key*, and return its current value. *Enumerate* is to return each key-value pair in turn.

The NIS supplies match and enumerate operations in a network environment where high availability and reliability are required. It provides availability and reliability by replicating both databases and database servers on multiple nodes within a single local net, and within the Internet. The database is replicated. All changes are made at a single server and eventually propagate to the remaining servers without locking. The NIS is appropriate for an environment in which changes to the mapping databases occur on the order of tens per day.

The NIS operates on an arbitrary number of map databases. Map names provide the lower of two levels of a naming hierarchy. Maps are themselves grouped into named sets, called domains. Domain names provide a second, higher level of naming. Map names must be unique within a domain, but can be duplicated in different domains. The NIS client interface requires that both a map name and a domain name be supplied to perform match and enumeration operations.

The NIS achieves high availability by replication. One area not addressed by the protocol that must be addressed by the implementors is global consistency among the replicated copies of the database. Every implementation should be designed so that at steady state a request yields the same result when it is made of any NIS database server. Update and update-propagation mechanisms must be implemented to supply the required degree of consistency.

Remote Procedure Call (RPC)

The Remote Procedure Call (RPC) mechanism defines a paradigm for interprocess communication modeled on function calls. Clients call functions that optionally return values. All inputs and outputs to the functions are in the client's address space. The function is executed by a server program.

Using RPC, clients address servers by a program number (this identifies the application level protocol that the server speaks) and a version number. Additionally, each server procedure has a procedure number assigned to it.

In an Internet environment, clients must also know the server's host Internet address and the server's rendezvous port. The server listens for service requests at ports associated with a particular transport protocol TCP/IP or UDP/IP.

The format of the data structures used as inputs to and outputs from the remotely executed procedures are typically defined by header files that are included when the client interface functions are compiled. Levels above the client interface package need not know any particulars of the RPC interface to the server.

External Data Representation (XDR)

The External Data Representation (XDR) specification establishes standard representations for basic data types (such as strings, signed and unsigned integers, and structures and unions) in a way that allows them to be transferred among machines with varying architectures.

XDR provides primitives to encode (that is, translate from the local host's representation to the standard representation) and decode (translate from the standard representation to the local host's representation) basic data types.

Constructor primitives allow arbitrarily complex data types to be made from the basic types.

The NIS's RPC input and output data structures are described using XDR's data definition language. In general, the data definition language looks like the C language, with a few extra constructs. One such extra construct is the discriminated union. This is like a C language union, in that it can hold various objects, but differs from it in that a discriminant indicates which object it currently holds. The discriminant is the first thing across the wire. Consider this simple example:

```
union switch (long int) {
   1:
      string exmpl_name<16>
   0:
      unsigned int exmpl_error_code
default:
   struct {}
}
```

The example should be interpreted as follows: the first object to be encoded/decoded (that is, the discriminant) is a long integer. If it has the value one, the next object is a string. If the discriminant has the value zero, the next object is an unsigned integer. If the discriminant takes any other value, don't encode or decode any more data.

A string data type in the XDR data definition language adds the ability to specify the maximum number of elements in an byte array or string of potentially variable size. For instance:

```
string domain<YPMAXDOMAIN>;
```

states that the byte sequence may be less than or equal to YPMAXDOMAIN bytes long.

An additional primitive data type is a boolean, which takes the value one to mean TRUE and zero to mean FALSE.

Maps and map operations

Map structure

Maps are named sets of key-value pairs. Keys and their values are counted binary objects, and may be ASCII information, but need not be. The data comprising a map is determined by the client applications that are the final customers for the data, not by the NIS. NIS has no syntactic nor semantic knowledge of the map contents. NIS doesn't determine or know any map's name. Map names are managed by the NIS clients. Conflict in the map namespace must be resolved by human administrators outside the NIS system.

Typical implementations for NIS maps are files or database management systems. The design of the NIS map database is an implementation detail and is unspecified by the protocol.

Match operation

NIS supports an exact match operation in the `YPPROC_MATCH` procedure. That is, if a match string and some key in the map are exactly the same, the value of the key is returned. No pattern matching, case conversion, or wildcarding is supported.

Map entry enumeration

It is possible to get the first key-value pair in a map with `YPPROC_FIRST`, and the next key-value pair with `YPPROC_NEXT`, calling "get first" once and "get next" until the return value indicates there are no more entries in the map. Making the same calls on the same map at the same NIS database server enumerates all entries in the same order. The actual order, however, is unspecified. Enumerating a map at a different NIS database server does not necessarily return entries in the same order.

Entire map retrieval

The `YPPROC_ALL` operation retrieves all key-value pairs in a map, with a single RPC request. This is faster than map entry enumeration, and more reliable, since it uses TCP. Ordering is the same as when enumeration is applied.

Map update

The update of NIS maps is an implementation detail that's outside the specification of the NIS service. A separate NIS update is used to achieve very limited update capability to NIS maps for Secure NFS.

Master and slave NIS database servers

For each map there is one NIS database server called the map's master. Map updates take place only on the master. An updated map should be transferred from the master to the rest of the NIS database servers, which are slave servers for this map.

It is possible for each map to have a different NIS database server as its master, or for all maps to have the same master, or any other combination. The choice of how to set up map masters is one of implementation and administrative policy.

Map propagation and consistency

Getting map updates from the master to the slaves is called map propagation. Neither technology nor algorithms for map propagation are specified by the protocol. Map propagation may be entirely manual. For instance, a person could copy the maps from the master to the slaves at a regular interval, or when a change is made on the master. This is unnecessarily labor intensive.

In order to escape from the idiosyncrasies of any particular implementation, all maps should be uniformly timestamped.

Functions to aid in map propagation

The way a map is transferred from one server to another is not specified by the NIS protocol. One possibility is for the system administrator to do it manually. Another is for the NIS database server to activate another process to perform the map transfer. A third is for a server to enumerate a recent version of the map, using the normal client map enumeration functions.

The `YPPROC_XFR` procedure requests the NIS server to update a map, and permits the actual transfer agent (some server process) to call back the requester with a summary status.

Domains

Domains provide a second level for naming within the NIS subsystem. They are names for sets of maps; and, therefore, create separate map name spaces. Domains provide an opportunity to break large organizations up into administrable chunks, and the ability to create parallel, non-interfering test and production environments.

Ideally, the domain of interest to a client ought to be associated with the invoking user, but in practice it is useful for client machines to be in a default domain. Implementations of the NIS client interface should supply some mechanism for telling processes the domain name they should use. This is needed not only because the concept of domain is a useless one as far as most programs are concerned, but, more importantly, so that programs can be written that are insensitive to both location and the invoking user.

Restrictions

The following capabilities are not included in the current NIS protocols.

Map update within the NIS

All write and delete accesses to the NIS map database are assumed to be outside of the NIS subsystem. It is probable that write access to the map database will be included in later versions of the NIS protocols. A limited form of NIS update protocol is provided for Secure NFS.

Version commitment across multiple requests

The NIS protocol was designed to keep the NIS database server stateless with regard to its clients. Therefore, there is no facility for contracting with a server to preallocate any resource beyond that required to service any single request. In particular, there is no way to get a server to commit to use a single version of a map while trying to enumerate that map's entries. Use `YPPROC_ALL` to avoid these problems.

Guaranteed global consistency

There is no facility for locking maps during the update or propagation phases, therefore it is virtually guaranteed that the map database be globally inconsistent during those phases. The set of client applications for which the NIS is an appropriate lookup service is one that (by definition) must be tolerant of transient inconsistencies.

Access control

The NIS database servers make no attempt to restrict access to the map data by any means. All syntactically correct requests are serviced.

NIS database server protocol definition

This section describes version 2 of the protocol. It is likely that changes will be made to successive versions.

RPC constants

The following numbers are in decimal:

- NIS database server protocol program number
- Current NIS protocol version

Other manifest constants

The following numbers are in decimal:

- Total maximum size of key and value for any pair. (The absolute sizes of the key and value may divide this maximum arbitrarily.)
- Maximum number of characters in a domain name
- Maximum number of characters in a map name
- Maximum number of characters in a NIS host name

Remote procedure return values

This section presents the return status values returned by several of the NIS remote procedures. All numbers are in decimal.

ypstat:

```
typedef enum {
   YP_TRUE  =  1,        /* General purpose success code. */
   YP_NOMORE  =  2,      /* No more entries in map. */
   YP_FALSE =  0,        /* General purpose failure code.*/
   YP_NOMAP = -1,        /* No such map in domain. */
   YP_NODOM = -2,        /* Domain not supported. */
   YP_NOKEY = -3,        /* No such key in map. */
   YP_BADOP = -4,        /* Invalid operation. */
   YP_BADDB = -5,        /* Server database is bad. */
   YP_YPERR = -6,        /* NIS server error. */
   YP_BADARGS = -7,      /* Request arguments bad. */
   YP_VERS  = -8         /* NIS server version mismatch. */
} ypstat
```

ypxfrstat:

```
typedef enum {
YPXFR_SUCC = 1,          /* Success */
YPXFR_AGE  = 2,          /* Master's version not newer */
YPXFR_NOMAP  =  -1,      /* Can't find server for map */
YPXFR_NODOM  =  -2,      /* Domain not supported */
YPXFR_RSRC =  -3,        /* Local resource alloc failure */
YPXFR_RPC  =  -4,        /* RPC failure talking to server */
YPXFR_MADDR  =  -5,      /* Can't get master address */
YPXFR_YPERR  =  -6,      /* NIS server/map db error */
YPXFR_BADARGS=  -7,      /* Request arguments bad */
YPXFR_DBM  =  -8,        /* Local database failure */
YPXFR_FILE =  -9,        /* Local file I/O failure */
YPXFR_SKEW = -10,        /* Map version skew in transfer */
YPXFR_CLEAR  = -11,      /* Can't clear local ypserv */
YPXFR_FORCE  = -12,      /* Must override defaults */
YPXFR_XFRERR = -13,      /* ypxfr error */
YPXFR_REFUSED= -14       /* ypserv refused transfer */
} ypxfrstat
```

Basic data structures

This section defines the data structures used as inputs to and outputs from the NIS remote procedures.

- domainname:

```
typedef string domainname<YPMAXDOMAIN>
```

- mapname:

```
typedef string mapname<YPMAXMAP>
```

- peername:

```
typedef string peername<YPMAXPEER>
```

- keydat:

```
typedef string keydat<YPMAXRECORD>
```

- valdat:

```
typedef string valdat<YPMAXRECORD>
```

- ypmap_parms:

```
typedef struct {
   domainname
   mapname
   unsigned long ordernum
   peername
} ypmap_parms
```

This contains parameters giving information about map `mapname` within domain domainname. `Peername` is the name of the map's master NIS database server. If any of the three strings is null, it indicates that information is unknown or unavailable. The ordernum element contains a binary value representing the value of the map's order number. If this is unavailable, `ordernum` is 0.

- ypreq_xfr:

```
typedef struct {
   struct ypmap_parms map_parms
   unsigned long transid
   unsigned long prog
   unsigned short port
} ypreq_xfr
```

- ypresp_val:

```
typedef struct {
   ypstat
   valdat
} ypresp_val
```

- ypresp_key_val:

```
typedef struct {
   ypstat
   keydat
   valdat
} ypresp_key_val
```

- ypresp_master:

```
typedef struct {
   ypstat
   peername
} ypresp_master
```

- ypresp_order:

```
typedef struct {
   ypstat
   unsigned long ordernum
} ypresp_order
```

- ypresp_all:

```
typedef union switch (boolean more) {
   TRUE:
   ypresp_key_val
   FALSE:
   struct {}
} ypresp_all
```

- ypresp_xfr:

```
typedef struct {
   unsigned long transid
   ypxfrstat
   xfrstat
} ypresp_xfr
```

- ypmaplist:

```
typedef struct {
   mapname
   ypmaplist *
} ypmaplist
```

- ypresp_maplist:

```
typedef struct {
   ypstat
   ypmaplist *
} ypresp_maplist
```

NIS database server remote procedures

This section contains a specification for each function that can be called as a remote procedure in Version 2. The input and output parameters are described using the XDR data definition language.

Procedure 0: Do Nothing

```
YPPROC_NULL() returns ()
```

This takes no arguments, does no work, and returns nothing. It is made available in all RPC services to allow server response testing and timing.

Procedure 1: Do You Serve This Domain?

```
YPPROC_DOMAIN (domain) returns (serves)
   domainname domain;
   boolean serves;
```

This returns TRUE if the server serves domain, and FALSE otherwise. This procedure allows a potential client to determine if a given server supports a certain domain.

Procedure 2: Answer Only If You Serve This Domain

```
YPPROC_DOMAIN_NONACK (domain) returns
      (serves)
   domainname domain;
   boolean serves;
```

This procedure returns TRUE if the server serves domain; otherwise, it does not return anything. The intent of the function is that it be called in a broadcast environment in which it is useful to restrict the number of useless messages. If this function is called, the client interface implementation must be written so as to regain control in the negative case: for instance, by means of a time-out on the response.

The current implementation currently returns the FALSE case by forcing an RPC decode error.

Procedure 3: Return Value of a Key

```
YPPROC_MATCH (req) returns (resp)
   ypreq_key req;
   ypresp_val resp;
```

This returns the value associated with the datum keydat in `req`. If the status element in `resp` has the value `YP_TRUE`, the value data are returned in the datum `valdat`.

Procedure 4: Get First Key-Value Pair in Map

```
PPROC_FIRST (req) returns (resp)
   ypreq_key req;
   ypresp_key_val resp;
```

If status has the value YP_TRUE, this returns the first key-value pair from the map named in req to the keydat and valdat elements within resp. When status contains the value YP_NOMORE, the map is empty.

Procedure 5: Get Next Key-Value Pair in Map

```
YPPROC_NEXT (req) returns (resp)
   ypreq_key req;
   ypresp_key_val resp;
```

If status has the value YP_TRUE, this returns the key-value pair following the key-value named req to the keydat and valdat elements within resp. If the passed key is the last key in the map, the value of status is YP_NOMORE.

Procedure 6: Transfer Map

```
YPPROC_XFR (req) returns (resp)
   ypreq_xfr req;
   ypresp_xfr resp;
```

The action taken in response to this request is unspecified, and is implementation dependent. The intention is to indicate to the server that a map should be updated, and to allow the actual transfer agent (whether it is the NIS server process, or some other process) to call back the requester with a summary status.

The transfer agent should call back the program running on the requesting host with program number `req.prog`, program version 1, and listening at port `req.port`. The procedure number is 1, and the callback data is of type `ypresp_xfr`. The `transid` field should turn around `req.transid`, and the `xfrstat` field should be set appropriately.

Procedure 7: Reinitialize Internal State

```
YPPROC_CLEAR () returns ()
```

The action taken in response to this request is unspecified and is implementation dependent. Different server implementations can have different amounts of internal state (open files, or the current map, for example). This request signals that all such state should be expunged.

Procedure 8: Get All Key-Value Pairs in Map

```
YPPROC_ALL (req) returns (resp)
   ypreq_nokey req;
   ypresp_all resp;
```

This allows all key-value pairs from a map to be transferred with a single RPC request. When the union's discriminant is FALSE, no more key-value pairs are returned. The status field of the last `rpresp_key_val` structure should be consulted to determine why the flow of returned key-value pairs has stopped.

Procedure 9: Get Map Master Name

```
YPPROC_MASTER (req) returns (resp)
   ypreq_nokey req;
   ypresp_master resp;
```

This returns the map's master NIS server inside the `resp` structure.

Procedure 10: Get Map Order Number

```
YPPROC_ORDER (req) returns (resp)
   ypreq_nokey req;
   ypresp_order resp;
```

This returns a map's order number as an unsigned long integer, which indicates when the map was built. This quantity represents the number of seconds since the midnight before January 1, 1970, Greewich Mean Time.

Procedure 11: Get All Maps in Domain

```
YPPROC_MAPLIST (req) returns (resp)
   domainname req;
   ypresp_maplist resp;
```

This returns a list of all the maps in a domain.

NIS binders

In order that any network service be usable, there must be some way for potential clients to find the servers. This chapter describes the NIS binder, an optional element in the NIS subsystem that supplies NIS database server addressing information to potential NIS clients.

In order to address a NIS server in the Internet environment, a client must know the server's Internet address, and the port at which the server is listening for service requests. No contract is negotiated between a NIS server and a potential client; therefore, the addressing information is sufficient to bind the client to the server.

Of the many possible ways for a client to get the addressing information, one alternative is to supply an entity to cache the bindings and to serve that binding database to potential NIS clients. The theory is that if finding the service takes a lot of work, you should allocate a specialist to do it, rather than burden every client with a job that is irrelevant to its intended function. A NIS binder is efficient only if it is easier for a client to find the NIS binder than to find a NIS database server, and if the NIS binder can itself find a NIS database server.

We make the assumption that a NIS binder is present at every network node, and because of this, addressing the NIS binder is easier than addressing a NIS database server. The scheme for finding a local resource is implementation-specific, but given that a resource is guaranteed to be local, there may be some efficient way of finding it. We further assume that the NIS binder can find a NIS database server somehow, but that the way is either complicated, time-consuming, or resource-consuming. If either of these assumptions is untrue, then your implementation is probably not a good bet for a NIS binder.

If a NIS binder is implemented, it can provide added value beyond the binding. It can verify that the binding is correct, and the NIS database server is alive and well, for instance. The degree of sureness in a binding that the NIS binder gives to a client is a parameter that can be tuned appropriately in the implementation.

NIS binder protocol definition

This section describes version 2 of the protocol. It is likely that changes will be made to successive versions.

RPC constants

All numbers are decimal.

- The NIS binder protocol program number
- The current NIS binder protocol version

Other manifest constants

All numbers are decimal.

- The maximum number of characters in a domain name

This is identical to the constant defined above within the NIS database server protocol section.

- `ypbind_resptype`:

```
enum ypbind_resptype {
   YPBIND_SUCC_VAL = 1,
   YPBIND_FAIL_VAL = 2
}
```

This discriminates between success responses and failure responses to a `YPBINDPROC_DOMAIN` request.

- `ypbinderr`:

```
typedef enum {
   YPBIND_ERR_ERR 1  /* Internal error */
   YPBIND_ERR_NOSERV 2 /* No bound server
                        * for domain */
   YPBIND_ERR_RESC 3 /* Can't allocate
                      * system resource */
} ypbinderr
```

The error case of most interest to a NIS binder client is `YPBIND_ERR_NOSERV`. It indicates that the binding request cannot be satisfied because the NIS binder doesn't know how to address any NIS database server in the named domain.

Basic data structures

This section defines the data structures used as inputs to and outputs from the NIS binder remote procedures.

- `domainname`:

```
typedef string domainname<YPMAXDOMAIN>
```

This is identical to the domainname string defined above within the NIS database server protocol section.

- `ypbind_binding`:

```
typedef struct {
   unsigned long ypbind_binding_addr
   unsigned short ypbind_binding_port
} ypbind_binding
```

This contains the information necessary to bind a client to a NIS database server in the Internet environment: `ypbind_binding_addr` holds the host IP address (4 bytes), and `ypbind_binding_port` holds the port address (2 bytes). Both IP address and port address must be in ARPA network byte order (most significant byte first, or big endian), regardless of the host machine's native architecture.

- `ypbind_resp`:

```
typedef struct {
   union switch (enum ypbind_resptype
                 status) {
      YPBIND_SUCC_VAL:
         ypbind_binding
      YPBIND_FAIL_VAL:
         ypbinderr
      default:{ }
   }
} ypbind_resp
```

This is the response to a `YPBINDPROC_DOMAIN` request.

- `ypbind_setdom`:

```
typedef struct {
   domainname
   ypbind_binding
   version
} ypbind_setdom
```

This is the input data structure for the `YPBINDPROC_SETDOM` procedure.

NIS binder remote procedures

Like the NIS procedures earlier, these procedures are described using the XDR data definition language.

Procedure 0: Do Nothing

```
YPBINDPROC_NULL () returns ()
```

This does no work. It is made available in all RPC services to allow server response testing and timing.

Procedure 1: Get Current Binding for a Domain

```
YPBINDPROC_DOMAIN (domain) returns (resp)
   domainname domain;
   ypbind_resp resp;
```

This returns the binding information necessary to address a NIS database server within the Internet environment.

Procedure 2: Set Domain Binding

```
YPBINDPROC_SETDOM (setdom) returns ()
   ypbind_setdom setdom;
```

This instructs a NIS binder to use the passed information as its current binding information for the passed domain.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

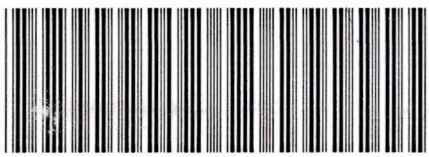
W

X

Y

ORDER NUMBER
DSW-106

DOCUMENT NUMBER
710-023730-000



CONVEX PRESS