Tru64 UNIX

Programming with ONC RPC

Part Number: AA-RH9XB-TE

July 1999

Product Version: Tru64 UNIX Version 5.0 or higher

Operating System and Version:

High-level programming through open-network remote procedure calls (ONC RPC) provides client-to-server communication for network application development without the need to program most of the interface to the underlying network.

Hewlett-Packard Company
Palo Alto, California

Contents

About This Manual

This manual provides an overview of high-level programming with open network computing remote procedure calls (ONC RPC), describes how to use the `rpcgen` protocol compiler to create applications, and describes the RPC programming interface. See `rpcgen`(1) for more information.

Audience

This guide assumes a knowledge of network theory and is for experienced programmers who want to write network applications using open-network computing remote procedure calls (ONC RPC) without needing to know about the underlying network.

Organization

This guide contains three chapters, an appendix, and an index:

Chapter 1	Provides an overview of high-level programming through remote procedure calls (RPC), and discusses the RPC model and versions, external data representation, and RPC independence from network transport protocol. This chapter is for anyone interested in ONC RPC.
Chapter 2	Describes how to write RPC client and server applications with the `rpcgen` protocol compiler. It also provides some information on `rpcgen`, client and server programming, debugging applications, the C preprocessor, and RPC language syntax. This chapter also describes how to create routines for external data representation (XDR). This chapter is for programmers who want to use `rpcgen` to write RPC-based network applications.
Chapter 3	Describes the RPC programming interface layers, XDR serialization defaults, raw RPC, and miscellaneous RPC features. This chapter is for programmers who need to understand RPC mechanisms to write customized network applications.
Appendix A	Contains additional notes on the XDR library. This appendix is for programmers who want to implement RPC and XDR on new machines.

New and Changed Information

The contents of this manual have not changed since the previous release.

Related Documents

The *Network Programmer's Guide* explains how to write programs that use networking system calls.

The *Technical Overview* provides general information about addressing, naming, routing, and network protocols and provides a summary of network applications native to HP Tru64 UNIX.

The *Network Administration: Connections* manual discusses network connection setup tasks and the day-to-day administration tasks, and explains how to isolate and correct problems with network connections.

The *Network Administration: Services* manual discusses network services setup tasks and day-to-day administration tasks, and explains how to isolate and correct problems with network services.

Icons on Tru64 UNIX Printed Manuals

The printed version of the Tru64 UNIX documentation uses letter icons on the spines of the manuals to help specific audiences quickly find the manuals that meet their needs. (You can order the printed documentation from HP.) The following list describes this convention:

G	Manuals for general users
S	Manuals for system and network administrators
P	Manuals for programmers
R	Manuals for reference page users

Some manuals in the documentation help meet the needs of several audiences. For example, the information in some system manuals is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the manuals in the Tru64 UNIX documentation set.

Reader's Comments

HP welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

A Reader's Comment form is located on your system in the following location:

`/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate HP technical support office. Information provided with the software media explains how to send problem reports to HP.

Conventions

This document uses the following conventions:

`%` `$`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.
`%` **`cat`**	Boldface type in interactive examples indicates typed user input.
`file`	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|] { | }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.

`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

1 Introduction to Remote Procedure Calls

High-level programming through remote procedure calls (RPC) provides logical client-to-server communication for network application development — without the need to program most of the interface to the underlying network. With RPC, the client makes a **remote procedure call** that sends requests to the server, which calls a dispatch routine, performs the requested service, and sends back a reply before the call returns to the client.

RPC does not require the caller to know about the underlying network. For example, a program can simply call `rnusers` (a C routine that returns the number of users on a remote machine) much like making a system call to `malloc`. You can make remote procedure calls from any language, and between different processes on the same machine.

1.1 The RPC Model

The remote procedure call model is similar to that of the local model, which works as follows:

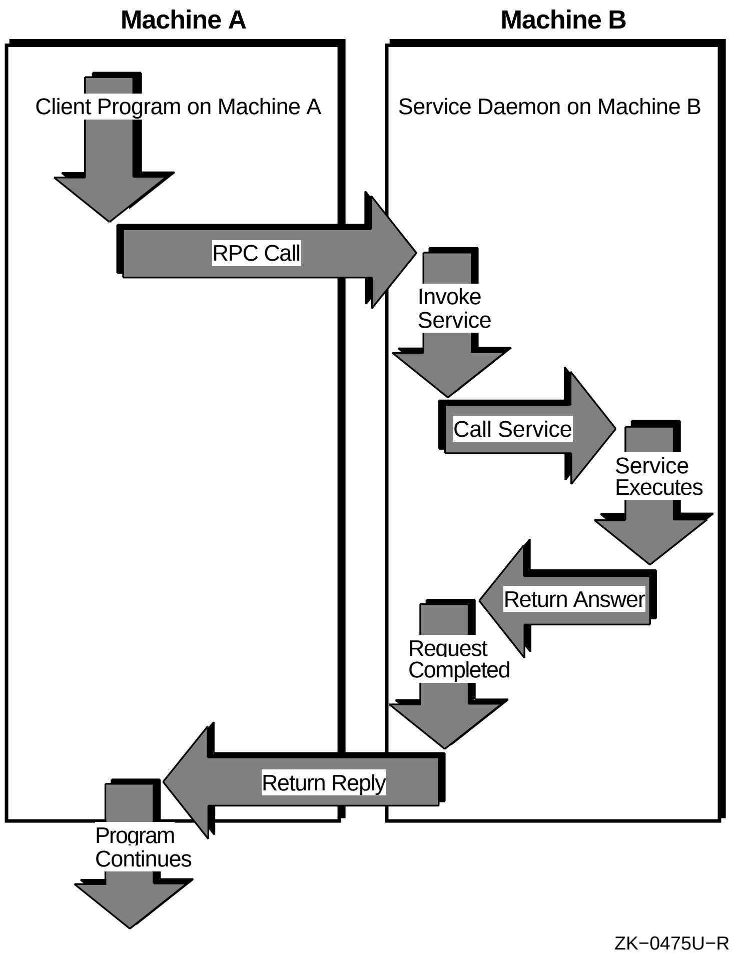
1. The caller places arguments to a procedure in a specific location (such as a result register).
2. The caller temporarily transfers control to the procedure.
3. When the caller gains control again, it obtains the results of the procedure from the specified location.
4. The caller then continues program execution.

The remote procedure call is similar, in that one thread of control logically winds through two processes — that of the caller and that of the server:

1. The caller process sends a call message to the server process and blocks (that is, waits) for a reply message. The **call message** contains the parameters of the procedure and the **reply message** contains the procedure results.
2. When the caller receives the reply message, it gets the results of the procedure.
3. The caller process then continues executing.

On the server side, a process is dormant — awaiting the arrival of a call message. When one arrives, the server process computes a reply that it then sends back to the requesting client. After this, the server process becomes dormant again. Figure 1–1 illustrates this basic form of network communication with the remote procedure call.

Figure 1–1: Basic Network Communication with Remote Procedure Call



This figure shows a synchronous RPC call, in which only one of the two processes is active at a given time. The remote procedure call hides the details of the network transport. However, the RPC protocol does not restrict the concurrency model. For example, RPC calls may be asynchronous so that the client can do another task while waiting for the reply from the server. Another possibility is that the server could create a task to process a certain type of request automatically, freeing it to service other requests.

Although RPC provides a way to avoid programming the underlying network transport, it still allows this where necessary.

1.2 RPC Procedure Versions

Each RPC procedure is uniquely defined by program and procedure numbers. 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 that, when a minor change is made to a remote service (adding a new procedure, for example), a new program number does not have to be assigned. When you want to call a procedure to find the number of remote users, you must know the appropriate program, version, and procedure numbers to use to contact the service. This information can be found in several sources. For example, the `/etc/rpc` file lists some RPC programs and the `rpcinfo` command lists the registered RPC programs and corresponding version numbers running on a particular system.

Typically, a service provides a protocol description so that you can write client applications that call the service. The RPC Administrator at Sun Microsystems, Inc. has a list of programs that have been registered with them (that is, have received port numbers from them) but you can write your own local RPC programs. Knowing the program and procedure numbers is useful only if the program is running on a system that you have access to.

1.3 Using portmap to Determine the Destination Port Number of RPC Packets

The `portmap` network service command starts automatically when a machine is booted. As part of its initialization, a server program calls its host portmap to create a portmap entry for its program and version number. To find the port of a remote program, a client sends an RPC call message to a server portmap. If the remote program is registered with the portmap, it returns the relevant port number in an RPC reply message. The client program can then send RPC call message packets to that remote program port.

The `portmap` network service has a well-known (dedicated) port. Other network service port numbers can be assigned statically or dynamically when they register their ports with the portmap of their host. Refer to the `portmap`(8) reference page for more information about the port mapping service.

1.4 RPC Independence from Transport Protocol

The RPC protocol is concerned only with the specification and interpretation of messages; it is independent of transport protocols because it needs no information on how a message is passed among processes.

Also, RPC does not implement any kind of reliability — the application itself must be aware of the transport protocol type underlying RPC. With a reliable transport, such as TCP/IP, the application need not do much else. However, an application must use its own retransmission and time-out policy if it is running on top of an unreliable transport, such as UDP/IP.

Because of transport independence, the RPC protocol does not actively interpret anything about remote procedures or their execution. Instead, the application infers required information from the underlying protocol (where such information should be explicitly specified). For example, if RPC is running on top of an unreliable transport (such as UDP/IP), and the application retransmits RPC messages after short time-outs, and if the application receives no reply, then it can infer only that a certain procedure was executed zero or more times. If it receives a reply, then the application infers that the procedure was executed at least once.

With 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

Even with a connection-oriented protocol like TCP, an application still needs time-outs and reconnection to handle server crashes.

ONC RPC is currently supported on both UDP/IP and TCP/IP transports. The selection of the transport depends on the application requirements. The UDP transport, which is connectionless, is a good choice if the application has the following characteristics:

- The procedures are idempotent; that is, the same procedure can be executed more than once without any side effects. For example, reading a block of data is idempotent; creating a file is not.
- The size of both the arguments and results is smaller than the UDP packet size of 8K bytes.
- The server is required to handle as many as several hundred clients. The UDP server can do so because it does not retain any information about the client state. By contrast, the TCP server holds state information for each open client connection and this limits its available resources.

TCP (connection-oriented) is a good transport choice if the application has any of the following characteristics:

- The application needs a reliable underlying transport.
- The procedures are non-idempotent.
- The size of either the arguments or the results exceeds 8K bytes.

1.5 External Data Representation (XDR)

RPC can handle arbitrary data structures, regardless of the byte order or structure layout convention on a machine. It does this by converting them to a network standard called External Data Representation (XDR) before sending them over the wire. XDR is a machine-independent description and encoding of data that can communicate between diverse machines, such as a VAX, Sun workstation, IBM-PC, or Cray.

Converting from a particular machine representation to XDR format is called **serializing**; the reverse process is **deserializing**.

1.6 Using rpcinfo to Get RPC Registration Information

The `rpcinfo` command reports current RPC registration information known to `portmap`; administrators can use it to delete registrations. The `rpcinfo` command can also find the RPC services registered on a specific host and report their port numbers and the transports for which the services are registered. You can also use it to call (through `ping`) a program version on a specific host using the TCP or UDP transport, and to report whether the response was received. For more information, see the `rpcinfo`(8) reference page.

1.7 Assigning Program Numbers

Program numbers are assigned in groups of `0x20000000` according to the following chart:

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 range of numbers, which should be identical for all ONC users. An ONC RPC application for general use should have an assigned number in this first range. The second range of numbers is for specific, user-defined customer applications, and is primarily for debugging new programs. The third, called the `Transient` group, is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and are not used.

To register a protocol specification, send a request by network mail to `rpc@sun.com` or write to:

RPC Administrator
Sun Microsystems
2550 Garcia Ave.
Mountain View, CA 94043

Include a compilable `rpcgen .x` file describing your protocol; you will then receive a unique program number. See `rpcgen`(1) for more information. Some of the RPC program numbers are in `/etc/rpc`.

2 Writing RPC Applications with the rpcgen Protocol Compiler

The `rpcgen`protocol compiler accepts a remote program interface definition written in RPC language, which is similar to C. It then produces C language output consisting of skeleton versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, a header file that contains common definitions, and optionally, dispatch tables that the server uses to invoke routines that are based on authorization checks. See `rpcgen`(1)for more information.

The **client skeleton** interface to the RPC library hides the network from its callers, and the **server skeleton** hides the network from the server procedures invoked by remote clients. You compile and link output files from `rpcgen` as usual. The server code generated by `rpcgen` supports `inetd`. You can start the server via `inetd` or at the command line.

You can write server procedures in any language that has system calling conventions. To get an executable server program, link the server procedure with the server skeleton from `rpcgen`. To create an executable program for a remote program, write an ordinary main program that makes local procedure calls to the client skeletons, and link the program with the `rpcgen` skeletons. If necessary, the `rpcgen` options enable you to suppress skeleton generation and specify the transport to be used by the server skeleton.

The `rpcgen` protocol compiler helps to reduce development time in the following ways:

- It greatly reduces network interface programming.
- It can mix low-level code with high-level code.
- For speed-critical applications, you can link customized high-level code with the `rpcgen` output.
- You can use `rpcgen` output as a starting point, and rewrite as necessary.

Refer to `rpcgen`(1) for more information about programming applications that use remote procedure calls or for writing XDR routines that convert procedure arguments and results into their network format (or vice versa). For a discussion of RPC programming without `rpcgen`, see Chapter 3.

2.1 Simple Example: Using rpcgen to Generate Client and Server RPC Code

This section shows how to convert a simple routine — one that prints messages to the console of a single machine — to an ONC RPC application that runs remotely over the network. To do this, the `rpcgen` protocol compiler is used to generate client and server RPC code. Example 2–1 shows the routine before conversion.

Example 2–1: Printing a Remote Message Without ONC RPC

```
/* 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 showing result. */
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);
}
```

Compile and run this program from a machine called `cyrkle`:

```
cyrkle% cc printmsg.c -o printmsg
cyrkle% printmsg "red rubber ball"
Message delivered!
cyrkle%
```

If the `printmessage` procedure at the bottom of the `printmsg.c` program of Example 2–1 were converted into a remote procedure, you could call it from anywhere in the network, instead of only from the program where it is embedded. Before doing this, it is necessary to write a protocol specification in RPC language that describes the remote procedure, as shown in the next section.

2.1.1 RPC Protocol Specification File Describing Remote Procedure

To create the specification file, you must know all the input and output parameter types. In Example 2–1, the `printmessage` procedure takes a string as input, and returns an integer as output. Example 2–2 is the RPC protocol specification file that describes the remote version of the `printmessage` procedure.

Example 2–2: RPC Protocol Specification File, Simple Example

```
/*
 * msg.x: Remote message printing protocol
 */

program MESSAGEPROG {
        version MESSAGEVERS {
                int PRINTMESSAGE(string) = 1;
        } = 1;
} = 0x20000099;
```

Remote procedures are part of remote programs, so Example 2–2 actually declares a remote program containing a single procedure, `PRINTMESSAGE`. By convention, all RPC services provide for a NULL procedure (procedure 0), normally used for "pinging." (See `ping`(8).) The RPC protocol specification file in Example 2–2 declares the `PRINTMESSAGE` procedure to be in version 1 of the remote program. No NULL procedure (procedure 0) is necessary in the protocol definition because `rpcgen` generates it automatically and the user does not need it.

In RPC language, the convention (though not a requirement) is to make all declarations in uppercase characters. Notice that the argument type is `string`, not `char *`, because `char *` in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also be a pointer to a single character or to an array of characters. In RPC language, a null-terminated string is unambiguously of type `string`.

2.1.2 Implementing the Procedure Declared in the Protocol Specification

Example 2–3 defines the remote procedure declared in the RPC protocol specification file of the previous example.

Example 2–3: Remote Procedure Definition

```
/*
 * msg_proc.c: implementation of the remote procedure
 * "printmessage"
 */

#include <stdio.h>
#include <rpc/rpc.h>  /* always needed */
#include "msg.h"      /* msg.h will be generated by rpcgen */

/*
 * Remote version of "printmessage"
 */
int *
printmessage_1(msg)   1
        char **msg;       2
{
        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);    3
}
```

In this example, the declaration of the remote procedure, `printmessage_1`, differs from that of the local procedure `printmessage` in three ways:

1 It has _1 appended to its name. In general, all remote procedures called by `rpcgen` are named by the following rule: the name in the procedure definition (here, `PRINTMESSAGE`) is converted to all lowercase letters, and an underscore (_) and version number (here, 1) is appended to it.

2 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; if there are no arguments, specify `void`.

3 It returns a pointer to an integer instead of an integer itself. This is also characteristic of remote procedures — they return pointers to their results. Therefore it is important to have the result declared as a `static`; if there are no arguments, specify `void`.

2.1.3 The Client Program That Calls the Remote Procedure

Example 2–4 declares the main client program, `rprintmsg.c`, that will call the remote procedure.

Example 2–4: Client Program that Calls the Remote Procedure

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */

#include <stdio.h>
#include <rpc/rpc.h>     /* always needed  */
#include "msg.h"         /* 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);
        }

        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");           1 2
```

Example 2–4: Client Program that Calls the Remote Procedure (cont.)

```
if (cl == NULL) {

        /*
         * Couldn’t establish connection with server.
         * Print error message and stop.
         */

        clnt_pcreateerror(server);
        exit(1);
}

/*
 * Call the remote procedure "printmessage" on the server
 */

result = printmessage_1(&message, cl);  3
if (result == NULL) {                 4

        /*
         * An error occurred while calling the server.
         * Print error message and stop.
         */

        clnt_perror(cl, server);
        exit(1);
}

/*
 * Okay, we successfully called the remote procedure.
 */

if (*result == 0) {     4

        /*
         * Server was unable to print our message.
         * Print error message and stop.
         */

fprintf(stderr, "%s: %s couldn’t print your message\n",
        argv[0], server);
        exit(1);
}

/*
 * The message got printed on the server’s console
 */
```

Example 2–4: Client Program that Calls the Remote Procedure (cont.)

```
        printf("Message delivered to %s!\n", server);
        exit(0);
}
```

In this example, the following events occur:

1 A client "handle" is created first, using the RPC library routine `clnt_create()`. This client handle will be passed to the skeleton routines that call the remote procedure.

2 The last parameter to `clnt_create` is `tcp`, the transport on which you want to run your application. (Alternatively `udp` could have been used.)

3 The remote procedure `printmessage_1` is called exactly the same way as in `msg_proc.c`, except for the inserted client handle as the second argument.

4 The remote procedure call can fail in two ways: the RPC mechanism itself can fail or there can be an error in the execution of the remote procedure. In the former case, the remote procedure, `print_message_1`, returns with a NULL. In the latter case, error reporting is application-dependent. In this example, the error is reported via `*result`.

2.1.4 Running rpcgen

Use the `rpcgen` protocol compiler on the RPC protocol specification file, `msg.x`, (from Example 2–2) to generate client and server RPC code automatically:

```
cyrkle% rpcgen msg.x
```

Using `rpcgen` like this — without options — automatically creates the following files from the input file `msg.x`:

- A header file called `msg.h` that contains `#define` statements for `MESSAGEPROG`, `MESSAGEVERS`, and `PRINTMESSAGE` so that they can be used in the other modules. You must include `msg.h` in both the client and server modules.
- An output file and the client skeleton routines within it. The output file, `msg_clnt.c`, is formed by appending `_clnt` to the file name and substituting the file type suffix, `.c`. The `msg_clnt.c` file contains only one client skeleton routine, `printmessage_1`, referred from the `printmsg` client program.

- The server program, `msg_svc.c`, created by appending `_svc` to the file name and substituting the file type suffix, `.c`. The `msg_svc.c` program calls `printmessage_1` from `msg_proc.c`.

Note

The `-T` option of `rpcgen` creates an additional output file of index information for dispatching service routines.

2.1.5 Compiling the Client and Server Programs

After the `rpcgen` protocol compilation, use two `cc` compilation statements to create a client program and a server program:

- To create the client program called `rprintmsg`, compile the `rprintmsg.c` and `msg_clnt.c` programs together, and use the `-o` option to specify the executable output in the file `rprintmsg`:

```
cyrkle% cc rprintmsg.c msg_clnt.c -o rprintmsg
```

- To create a server program called `msg_server`, compile the `msg_proc.c` and `msg_svc.c` programs together, and use the `-o` option to specify the executable output in the file `msg_server`:

```
cyrkle% cc msg_proc.c msg_svc.c -o msg_server
```

2.1.6 Copying the Server to a Remote Machine and Running It

Copy the server program `msg_server` to a remote machine called `space` in this example. Then, run the program in background mode on the server—indicated by &:

```
space% msg_server &
```

Note

Servers generated by `rpcgen` can be invoked with port monitors like `inetd`, as well as from the command line, if they are invoked with the `-I` option.

From a local machine (`earth`) you can now print a message on the console of the remote machine `space`:

```
earth% rprintmsg space "Hello out there..."
```

The message `Hello out there...` will appear on the console of the machine `space`. You can print a message on any console (including your own) with this program if you copy the server to that machine and run it.

2.2 Advanced Example: Using rpcgen to Generate XDR Routines

Section 2.1 explains how to use `rpcgen` to generate client and server RPC code automatically to convert a simple procedure to one that can be run remotely over the network. The `rpcgen` protocol compiler can also generate the external data representation (XDR) routines that convert local data structures into network format (and vice versa).

The following sections present a more advanced example of a complete RPC service — a remote directory listing service that uses `rpcgen` to generate both skeleton and XDR routines.

2.2.1 The RPC Protocol Specification

As with the simple example, you must first create an RPC protocol specification file. This file, `dir.x`, is for a remote directory listing, shown in Example 2–5.

Example 2–5: RPC Protocol Specification File, Advanced Example

```
/*
 * dir.x: Remote directory listing protocol
 */

/* maximum length of a directory entry */
const MAXNAMELEN = 255;

/* a directory entry */
typedef string nametype<MAXNAMELEN>;

/* a link in the listing */
typedef struct namenode *namelist;

/*
 * 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 */
```

Example 2–5: RPC Protocol Specification File, Advanced Example (cont.)

```
default:
        void;           /* error occurred: nothing else to return */
};

/*
 * The directory program definition
 */
program DIRPROG {
        version DIRVERS {
                readdir_res
                READDIR(nametype) = 1;
        } = 1;
} = 0x20000076;
```

Note

You can define types (like `readdir_res` in Example 2–5) by using the `struct`, `union`, and `enum` keywords, but do not use these keywords in later variable declarations of those types. For example, if you define `union foo`, you must declare it later by using `foo`, not `union foo`. The `rpcgen` protocol compiler compiles RPC unions into C structures, so it is an error to declare them later by using the `union` keyword.

Running `rpcgen` on `dir.x` creates four output files:

- Header file
- File of client skeleton routines
- Server skeleton file
- File of XDR routines

The first three files have already been described. The fourth file, `dir_xdr.c`, contains the XDR routines that convert the declared data types into XDR format (and vice versa). For each data type used in the `.x` file, `rpcgen` assumes that the RPC/XDR library contains a routine with the name of that data type prefixed by `xdr_`, for example, `xdr_int`. If the data type was defined in the `.x` file, then `rpcgen` generates the required XDR routine. If there are no such data types, then the file (for example, `dir_xdr.c`) will not be generated. If the data types were used but not defined, then the user has to provide that XDR routine. This enables you to create your own customized XDR routines.

2.2.2 Implementing the Procedure Declared in the Protocol Specification

Example 2–6 consists of the `dir_proc.c` file that implements the remote `READDIR` procedure from the previous RPC protocol specification file.

Example 2–6: Remote Procedure Implementation

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>            /* Always needed */
#include <sys/dir.h>
#include "dir.h"                /* Created by rpcgen */

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);
        }

        /*
         * 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)) {
```

Example 2–6: Remote Procedure Implementation (cont.)

```
                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);
}
```

2.2.3 The Client Program that Calls the Remote Procedure

Example 2–7 shows the client side program, `rls.c`, that calls the remote server procedure.

Example 2–7: Client Program that Calls the Server

```
/*
 * 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);
        }
```

Example 2–7: Client Program that Calls the Server (cont.)

```
        server = argv[1];
        dir = argv[2];

        /*
         * Create client "handle" used for calling DIRPROG on
         * the server designated on the command line.  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 stop.
                 */
                clnt_pcreateerror(server);
                exit(1);
        }

        /*
         * Call the remote procedure readdir on the server
         */
        result = readdir_1(&dir, cl);
        if (result == NULL) {
                /*
                 * An RPC error occurred while calling the server.
                 * Print error message and stop.
                 */
                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 stop.
                 */
                errno = result->errno;
                perror(dir);
                exit(1);
        }

        /*
         * Successfully got a directory listing.
         * Print it out.
```

Example 2–7: Client Program that Calls the Server (cont.)

```
         */
        for (nl = result->readdir_res_u.list; nl != NULL;
          nl = nl->next) {
                printf("%s\n", nl->name);
        }
        exit(0);
}
```

2.2.4 Running rpcgen

As with the simple example, you must run the `rpcgen` protocol compiler on the RPC protocol specification file `dir.x`, to create a header file, `dir.h`, an output file of client skeletons, `dir_clnt.c`, and a server program, `dir_svc.c`. For this advanced example, `rpcgen` also generates the file of XDR routines, `dir_xdr.c`:

```
earth% rpcgen dir.x
```

2.2.5 Compiling the File of XDR Routines

The next step is to compile the file of XDR routines, `dir_xdr.c`, with the following `cc -c` command:

```
earth% cc -c dir_xdr.c
```

Here, the `-c` option is used to suppress the loading phase of the compilation and to produce an object file, even if only one program is compiled.

2.2.6 Compiling the Client Side Program with rpcgen Output

Next, the remote directory listing client program, `rls.c`, is compiled with the file of client skeletons, `dir_clnt.c`, and the object file of the previous compilation, `dir_xdr.o`. The `-o` option places the executable output of the compilation into the file `rls`:

```
earth% cc rls.c dir_clnt.c dir_xdr.o -o rls
```

2.2.7 Compiling the Server Side Program with rpcgen Output

The following statement compiles three files together: the server program from the original `rpcgen` compilation, `dir_svc.c`; the remote READDIR implementation program, `dir_proc.c`; and the object file, `dir_xdr.o`, produced by the recent `cc` compilation of the file of XDR routines. Through the `-o` option, the executable output is placed in the file `dir_svc`:

```
earth% cc dir_svc.c dir_proc.c dir_xdr.o -o dir_svc
```

2.2.8 Running the Remote Directory Program

To run the remote directory program, use the new `dir_svc` and `rls` commands. The following statement runs the `dir_svc` command in background mode on the local machine `earth`:

```
earth% dir_svc &
```

The `rls` command can then be used from the remote machine `space` to provide a directory listing on the machine where `dir_svc` is running in background mode. The command and output (a directory listing of `/usr/pub` on machine `earth`) is shown here:

```
space% rls earth /usr/pub
.
..
ascii
eqnchar
kbd
marg8
tabclr
tabs
tabs4
```

Note

Client code generated by `rpcgen` does not release the memory allocated for the results of the RPC call. You can call `xdr_free` to deallocate the memory when no longer needed. This is similar to calling `free`, except that you must also pass the XDR routine for the result.

For example, after printing the directory listing in the previous example, you could call xdr_free as follows:

```
xdr_free(xdr_readdir_res, result);
```

2.3 Debugging Applications

It is difficult to debug distributed applications that have separate client and server processes. To simplify this, you can test the client program and the server procedure as a single program by linking them with each other rather than with the client and server skeletons. To do this, you must first remove calls to client creation RPC library routines (for example, clnt_create). To create the single debuggable file rls, use the -o option when you compile rls.c, dir_clnt.c, dir_proc.c, and dir_xdr.c together as follows:

```
% cc rls.c dir_clnt.c dir_proc.c dir_xdr.c -o rls
```

The procedure calls are 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 skeleton produced by rpcgen and the server procedures can be linked to the server skeleton produced by rpcgen.

There are two kinds of errors possible in an RPC call:

- A problem with the remote procedure call mechanism

  This occurs when a procedure is unavailable, the remote server does not respond, the remote server cannot decode the arguments, and so on. As in Example 2–7, an RPC error occurs if result is NULL.

  The reason for the failure can be printed by using clnt_perror, or you can return an error string through clnt_sperror.

- A problem with the server itself

  As in Example 2–6, an error occurs if opendir fails; that is why readdir_res is of type union. The handling of these types of errors are the responsibility of the programmer.

2.4 The C-Preprocessor

The C-preprocessor, cpp, runs on all input files before they are compiled, so all the preprocessor directives are legal within an .x file. Five macro identifiers may have been defined, depending upon which output file is being generated. The following table lists these macros:

Identifier	Usage
RPC_HDR	For header-file output

RPC_XDR	For XDR routine output
RPC_SVC	For server-skeleton output
RPC_CLNT	For client-skeleton output
RPC_TBL	For index-table output

Also, `rpcgen` does some additional preprocessing of the input file. Any line that begins with a percent sign (%) passes directly into the output file, without any interpretation. Example 2–8 demonstrates this processing feature.

Example 2–8: Using the Percent Sign to Bypass Interpretation of a Line

```
/*
 * 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
```

Using the percent sign feature does not guarantee that `rpcgen` will place the output where you intend. If you have problems of this type, do not use this feature.

2.5 rpcgen Programming

The following sections contain additional `rpcgen` programming information about network types, defining symbols, `inetd` support, and dispatch tables.

2.5.1 Network Types

By default, `rpcgen` generates server code for both UDP and TCP transports. The `-s` flag creates a server that responds to requests on the specified transport. The following example creates a UDP server:

```
rpcgen -s udp proto.x
```

2.5.2 User-Provided Define Statements

The rpcgen protocol compiler provides a way to define symbols and assign values to them. These defined symbols are passed on to the C preprocessor when it is invoked. This facility is useful when, for example, invoking debugging code that is enabled only when the DEBUG symbol is defined. For example:

```
rpcgen -DDEBUG proto.x
```

2.5.3 inetd Support

The rpcgen protocol compiler can create RPC servers that can be invoked by inetd when a request for that service is received.

```
rpcgen -I proto.x
```

The server code in proto_svc.c supports inetd. For more information on setting up the entry for RPC services in /etc/inetd.conf, see Section 3.3.6.

In many applications, it is useful for services to wait after responding to a request, on the chance that another will soon follow. However, if there is no call within a certain time (by default, 120 seconds), the server exits and the port monitor continues to monitor requests for its services. You can use the -K option to change the default waiting time. In the following example, the server waits only 20 seconds before exiting:

```
rpcgen -I -K 20 proto.x
```

If you want the server to exit immediately, use -K 0; if you want the server to wait forever (a normal server situation), use -K -1.

2.5.4 Dispatch Tables

Dispatch tables are often useful. For example, the server dispatch routine may need to check authorization and then invoke the service routine, or a client library may need to control all details of storage management and XDR data conversion. The following rpcgen command generates RPC dispatch tables for each program defined in the protocol description file, proto.x, and places them in the file proto_tbl.i (the suffix .i indicates index):

```
rpcgen -T proto.x
```

Each entry in the table is a struct rpcgen_table defined in the header file, proto.h, as follows:

```
struct rpcgen_table {
        char        *(*proc)();
        xdrproc_t   xdr_arg;
        unsigned    len_arg;
        xdrproc_t   xdr_res;
        unsigned    len_res;
};
```

In this `proto.h` definition, `proc` is a pointer to the service routine, `xdr_arg` is a pointer to the input (argument) `xdr_routine`, `len_arg` is the length in bytes of the input argument, `xdr_res` is a pointer to the output (result) `xdr_routine`, and `len_res` is the length in bytes of the output result.

The table `dirprog_1_table` is indexed by procedure number. The variable `dirprog_1_nproc` contains the number of entries in the table. The `find_proc` routine in Example 2–9 shows how to locate a procedure in the dispatch tables.

Example 2–9: Locating a Procedure in a Dispatch Table

```
struct rpcgen_table *
find_proc(proc)
        long    proc;
{
        if (proc >= dirprog_1_nproc)

                /* error */
        else
                return (&dirprog_1_table[proc]);
}
```

Each entry in the dispatch table contains a pointer to the corresponding service routine. However, the service routine is not defined in the client code. To avoid generating unresolved external references, and to require only one source file for the dispatch table, the actual service routine initializer is `RPCGEN_ACTION(proc_ver)`. With this, you can include the same dispatch table in both the client and the server. Use the following `define` statement when compiling the client:

```
#define RPCGEN_ACTION(routine)  0
```

Next, use the following `define` statement when compiling the server:

```
#define RPCGEN_ACTION(routine)  routine
```

2.6 Client Programming

The following sections contain client programming information about default timeouts and client authentication.

2.6.1 Timeout Changes

RPC sets a default timeout of 25 seconds for RPC calls when `clnt_create` is used. RPC waits for 25 seconds to get the results from the server. If it does not, then this usually means one of the following conditions exists:

- The server is not running
- The remote machine has crashed
- The network is unreachable

In such cases, the function returns NULL; you can print the error with `clnt_perrno`.

Sometimes you may need to change the timeout value to accommodate the application or because the server is too slow or far away. Change the timeout by using `clnt_control`. The code segment in Example 2–10 demonstrates the use of `clnt_control`.

Example 2–10: Using the clnt_control Routine

```
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; /* this should always be set  */
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

2.6.2 Client Authentication

By default, client creation routines do not handle client authentication. Sometimes, you may want the client to authenticate itself to the server. This is easy to do, as shown in the following coding:

```
CLIENT *cl;

cl = client_create("somehost", SOMEPROG, SOMEVERS, "udp");
if (cl != NULL) {
        /* To set UNIX style authentication */
        cl->cl_auth = authunix_create_default();
}
```

For more information on authentication, see Section 3.3.5.

2.7 Server Programming

The following sections contain server programming information about system broadcasts and passing data to server procedures.

2.7.1 Handling Broadcasts

Sometimes, clients broadcast to determine whether a particular server exists on the network, or to determine all the servers for a particular program and version number. You make these calls with `clnt_broadcast` (for which there is no `rpcgen` support). Refer to Section 3.3.2.

When a procedure is known to be called via broadcast RPC, it is best for the server not to reply unless it can provide useful information to the client. Otherwise, the network can be overloaded with useless replies. To prevent the server from replying, a remote procedure can return NULL as its result; the server code generated by `rpcgen` can detect this and prevent a reply.

The following example shows a procedure that replies only if it acts as an NFS server:

```
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);
}
```

If a procedure returns type `void *`, it must return a non-NULL pointer if it wants RPC to reply for it.

2.7.2 Passing Data to Server Procedures

Server procedures often need to know more about an RPC call than just its arguments. For example, getting authentication information is useful to procedures that want to implement some level of security. This information is supplied to the server procedure as a second argument. (For details see the structure of `svc_req` in Section 3.3.5.2.) The following example shows the use of `svc_req`, where the previous `printmessage_1` procedure is rewritten to allow only root users to print a message to the console:

```
int *
printmessage_1(msg, rqstp)
```

```
        char **msg;
        struct svc_req  *rqstp;
{
        static int result;      /* Must be static */
        FILE *f;
        struct authunix_parms *aup;

        aup = (struct authunix_parms *)rqstp->rq_clntcred;
        if (aup->aup_uid != 0) {
                result = 0;
                return (&result);
        }

        /*
         * Same code as before.
         */
}
```

2.8 RPC and XDR Languages

RPC language is an extension of XDR language, through the addition of the `program` and `version` types. The XDR language is similar to C. For a complete description of the XDR language syntax, see RFC 1014, *External Data Representation Standard: Protocol Specification* . For a description of the RPC extensions to the XDR language, see RFC 1050, *Remote Procedure Calls: Protocol Specification* .

The following sections describe the syntax of the RPC and XDR languages, with examples and descriptions of how the various RPC and XDR type definitions are compiled into C type definitions in the output header file.

2.8.1 Definitions

An RPC language file consists of a series of definitions:

```
        definition-list:
                definition ";"
                definition ";" definition-list
```

RPC recognizes the following definition types:

```
        definition:
                enum-definition
                typedef-definition
                const-definition
                declaration-definition
                struct-definition
                union-definition
                program-definition
```

2.8.2 Enumerations

XDR enumerations have the same syntax as C enumerations:

```
        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
```

A comparison of the next two examples show how close XDR language is to C by showing an XDR language `enum` (Example 2–11), followed by the C language `enum` that results after compilation (Example 2–12).

Example 2–11: XDR enum Before Compilation

```
        enum colortype {
                RED = 0,
                GREEN = 1,
                BLUE = 2
        };
```

Example 2–12: C enum Resulting from Compiling XDR enum

```
        enum colortype {
                RED = 0,
                GREEN = 1,
                BLUE = 2,
        };
        typedef enum colortype colortype;
```

2.8.3 Typedefs

XDR typedefs have the same syntax as C typedefs:

```
        typedef-definition:
                "typedef" declaration
```

The following example in XDR defines a `fname_type` that declares file name strings with a maximum length of 255 characters:

```
typedef string fname_type<255>;
```

The following example shows the corresponding C definition for this:

```
typedef char *fname_type;
```

2.8.4 Constants

XDR constants are used wherever an integer constant is used (for example, in array size specifications), as shown by the following syntax:

```
const-definition:
        "const" const-ident "=" integer
```

The following XDR example defines a constant `DOZEN` equal to `12`:

```
const DOZEN = 12;
```

The following example shows the corresponding C definition for this:

```
#define DOZEN 12
```

2.8.5 Declarations

XDR provides only four kinds of declarations, shown by the following syntax:

```
declaration:
        simple-declaration        1
        fixed-array-declaration       2
        variable-array-declaration        3
        pointer-declaration           4
```

The syntax for each, followed by examples, is listed here:

1. Simple declarations

```
simple-declaration:
        type-ident variable-ident
```

   For example, `colortype color` in XDR, is the same in C: `colortype color`.

2. Fixed-length array declarations

```
fixed-array-declaration:
        type-ident variable-ident "[" value "]"
```

   For example, `colortype palette[8]` in XDR, is the same in C: `colortype palette[8]`.

3. Variable-length array declarations

   These have no explicit syntax in C, so XDR creates its own by using angle brackets, as in the following syntax:

```
variable-array-declaration:
        type-ident variable-ident "<" value ">"
        type-ident variable-ident "<" ">"
```

The maximum size is specified between the angle brackets; it may be omitted, indicating that the array can be of any size, as shown in the following example:

```
        int heights<12>;        /* at most 12 items */
        int widths<>;           /* any number of items */
```

Variable-length arrays have no explicit syntax in C, so each of their declarations is compiled into a `struct`. For example, the `heights` declaration is compiled into the following struct:

```
    struct {
        u_int heights_len; /* number of items in array */
        int *heights_val;  /* pointer to array */
    } heights;
```

Here, the `_len` component stores the number of items in the array and the `_val` component stores the pointer to the array. The first part of each of these component names is the same as the name of the declared XDR variable.

4 Pointer declarations

These are the same in XDR as in C. You cannot send pointers over the network, but you can use XDR pointers to send recursive data types, such as lists and trees. In XDR language, this type is called `optional-data`, not `pointer`, as in the following syntax:

```
optional-data:
        type-ident "*"variable-ident
```

An example of this (the same in both XDR and C) follows:

```
        listitem *next;
```

2.8.6 Structures

XDR declares a `struct` almost exactly like its C counterpart. The XDR syntax is 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, followed by the C structure into which it is compiled in the output header file:

```
struct coord {
        int x;
        int y;
};
```

The following example shows the C structure that results from compiling the previous XDR structure:

```
struct coord {
        int x;
        int y;
};
typedef struct coord coord;
```

Here, the output is identical to the input, except for the added `typedef` at the end of the output. This enables the use of `coord` instead of `struct coord` in declarations.

2.8.7 Unions

XDR unions are discriminated unions, and are different from C unions. They are more analogous to Pascal variant records than to C unions. The syntax is shown here:

```
union-definition:
        "union" union-ident "switch" ("simple declaration") "{"
                case-list
        "}"

case-list:
        "case" value ":" declaration ";"
        "case" value ":" declaration ";" case-list
        "default" ":" declaration ";"
```

The following is an example of a type that might be returned as the result of a `read data`. If there is no error, it returns a block of data; otherwise, it returns nothing:

```
union read_result switch (int errno) {
case 0:
        opaque data[1024];
default:
        void;
};
```

This coding is 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 same name as the structure type name, except for the suffix, `_u`.

2.8.8 Programs

You declare RPC programs 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
```

Example 2–13 shows a program of time protocol.

Example 2–13: RPC Program Illustrating Time Protocol

```
/*
 * 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 the following `#define` statements in the output header file:

```
#define TIMEPROG 44
#define TIMEVERS 1
#defRine TIMEGET 1
#define TIMESET 2
```

2.8.9 Special Cases

The following list describes exceptions to the syntax rules described in the previous sections:

- 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. Items declared as type `bool` in XDR language are compiled into `bool_t` in the output header file. For example, `bool married` is compiled into `bool_t married`.

- Strings

  C has no built-in string type, but instead uses the null-terminated `char *` convention. In XDR language, you declare strings by using the `string` keyword, and each string is compiled into a `char *` in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (excluding the NULL character). For example, `string name<32>` would be compiled into `char *name`. You can omit a maximum size to indicate a string of arbitrary length. For example, `string longname<>` would be compiled into `char *longname`.

- Opaque data

  RPC and XDR use **opaque data** to describe untyped data, which consists simply of sequences of arbitrary bytes. You declare opaque data as an array of either fixed or variable length. An `opaque` declaration of a fixed length array is `opaque diskblock[512]`, whose C counterpart is `char diskblock[512]`. An `opaque` declaration of a variable length array is `opaque filedata<1024>`, whose C counterpart could be the following:

  ```
          struct {
                  u_int filedata_len;
                  char *filedata_val;
          } filedata;
  ```

- Voids

  In a `void` declaration, the variable is not named. The declaration is just a `void`. Declarations of `void` occur only in union and program definitions (as the argument or result of a remote procedure).

3
RPC Programming Interface

For most applications, you do not need the information in this chapter; you can simply use the automatic features of the `rpcgen` protocol compiler (described in Chapter 2. This chapter requires an understanding of network theory; it is for programmers who must write customized network applications using open-network computing remote procedure calls (ONC RPC), and who need to know about the RPC mechanisms hidden by `rpcgen`.

3.1 RPC Layers

The ONC RPC interface consists of three layers, highest, middle, and lowest. For ONC RPC programming, only the middle and lowest layers are of interest; the highest layer is transparent to the operating system, machine, and network upon which it is run. For a complete specification of the routines in the remote procedure call library, see the `rpc`(3) reference page.

The middle layer routines are adequate for most applications. This layer is "RPC proper" because you do not need to write additional programming code for network sockets, the operating system, or any other low-level implementation mechanisms. At this level, you simply make remote procedure calls to routines on other machines. For example, you can make simple ONC RPC calls by using the following system routines:

- `registerrpc`, which obtains a unique system-wide procedure-identification number
- `callrpc`, which executes a remote procedure call
- `svc_run`, which calls a remote procedure in response to an RPC request

The middle layer is not suitable for complex programming tasks because it sacrifices flexibility for simplicity. Although it is adequate for many tasks, the middle layer does not enable the following:

- Timeout specifications
- Choice of transport
- Operating system process control
- Processing flexibility after occurrence of error
- Multiple kinds of call authentication

The lowest layer is suitable for programming tasks that require greater efficiency or flexibility. The lowest layer routines include client creation routines such as:

- `clnt_create`, which creates a client handle
- `clnt_call`, which calls the server `svcudp_create`, a server creation routine, and `svc_register`, the server registration routine

The following sections describe the middle and lowest RPC layers.

3.1.1 Middle Layer of RPC

The middle layer is the simplest RPC program interface; from this layer you make explicit RPC calls and use the functions `callrpc` and `registerrpc`.

3.1.1.1 Using callrpc

The simplest way to make remote procedure calls is through the RPC library routine `callrpc`. The programming code in Example 3–1, which obtains the number of remote users, shows the usage of `callrpc`.

Example 3–1: Using callrpc

```
/*
 * Print the number of users on a remote systedm using callrpc
 */

#include <stdio.h>
#include <rpc/rpc.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]);
```

Example 3–1: Using callrpc (cont.)

```
        exit(0);
}
```

The `callrpc` library routine has eight parameters. In Example 3–1 the first parameter, `argv[1]`, is the name of the remote server machine. The next three, `RUSERSPROG`, `RUSERSVERS`, and `RUSERSPROC_NUM`, are the program, version, and procedure numbers that together identify the procedure to be called. The fifth and sixth parameters are an XDR filter (`xdr_void`) and an argument (`0`) to be encoded and passed to the remote procedure. You provide an **XDR filter** procedure to encode or decode machine-dependent data to or from the XDR format.

The final two parameters are an XDR filter, `xdr_u_long`, for decoding the results returned by the remote procedure and a pointer, `&nusers`, to the storage location of the procedure results. Multiple arguments and results are handled by embedding them in structures.

If `callrpc` completes successfully, it returns zero; otherwise it returns a nonzero value. The return codes are found in `rpc/clnt.h`. The `callrpc` library routine needs the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). For `RUSERSPROC_NUM`, the return value is an `unsigned long`. This is why `callrpc` has `xdr_u_long` as its first return parameter, which means that the result is of type `unsigned long`, and `&nusers` as its second return parameter, which is a pointer to the location that stores the `long` result. `RUSERSPROC_NUM` takes no argument, so the argument parameter of `callrpc` is `xdr_void`. In such cases the argument must be NULL.

If `callrpc` gets no answer after trying several times to deliver a message, it returns with an error code. 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 in Section 3.1.2.

The remote server procedure corresponding to the `callrpc` usage example might look like the one in Example 3–2.

Example 3–2: Remote Server Procedure

```
unsigned long *
nuser(indata)
        char *indata;
{
        static unsigned long nusers;

        /*
```

Example 3–2: Remote Server Procedure (cont.)

```
         * Code here to compute the number of users
         * and place result in variable nusers.
         */
        return(&nusers);
}
```

This procedure takes one argument, a pointer to the input of the remote procedure call (ignored in the example), and returns a pointer to the result. In the current version of C, character pointers are the generic pointers, so the input argument and the return value can be cast to `char *`.

3.1.1.2 Using registerrpc

Normally, a server registers all of the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests. Using `rpcgen` for this also generates a server dispatch function. You can write a server yourself by using `registerrpc`. Example 3–3 is a program segment showing how you would use `registerrpc` in the main body of a server program that registers a single procedure; the remote procedure call passes a single `unsigned long`.

Example 3–3: Using registerrpc in the Main Body of a Server Program

```
#include <stdio.h>
#include <rpc/rpc.h>            /* required */
#include <rpcsvc/rusers.h>      /* for prog, vers definitions */

unsigned long *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 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 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.)

The underlying transport mechanism for `registerrpc` is UDP.

Note

The UDP transport mechanism can handle only arguments and results that are less than 8K bytes in length.

After registering the local procedure, the main procedure of the server program calls `svc_run`, the remote procedure dispatcher for the RPC library; `svc_run` calls the remote procedures in response to RPC requests and decodes remote procedure arguments and encodes results. To do this, it uses the XDR filters specified when the remote procedure was registered with `registerrpc`.

3.1.1.3 Passing Arbitrary Data Types

RPC can handle arbitrary data structures, regardless of machine conventions, for byte order and structure layout, by converting them to a network standard called External Data Representation (XDR) before sending them over the network. 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 the previous example), or one that you supply. XDR has the following built-in routines:

Built-in XDR primitive routines:

xdr_hyper	xdr_u_hyper	xdr_enum
xdr_int	xdr_u_int	xdr_bool
xdr_long	xdr_u_long	xdr_wrapstring
xdr_longlong_t	xdr_u_longlong_t	
xdr_short	xdr_u_short	
xdr_char	xdr_u_char	

Other built-in XDR routines:

xdr_array	xdr_bytes	xdr_reference
xdr_vector	xdr_union	xdr_pointer
xdr_string	xdr_opaque	

You cannot use the `xdr_string` routine with either `callrpc` or `registerrpc`, each of which passes only two parameters to an XDR routine.

Instead, use `xdr_wrapstring`, which takes only two parameters and calls `xdr_string`.

3.1.1.4 User-Defined Routines

Suppose that you want to send the following structure:

```
struct simple {
        int a;
        short b;
} simple;
```

To send it, you would use the following `callrpc` call:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
        xdr_simple, &simple ...);
```

With this call to `callrpc`, you could define the routine `xdr_simple` as in the following example:

```
#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 (evaluates to TRUE in C) if it completes successfully; otherwise, it returns zero: For a complete description of XDR, refer to RFC 1014, *XDR Protocol Specification* and Appendix A.

Note

It is best to use `rpcgen` to generate XDR routines. Use the `-c` option of `rpcgen` to generate only the `_xdr.c` file.

As another example, if you want to send a variable array of integers, you might package them as a structure like this:

```
struct varintarr {
        int *data;
        int arrlnth;
} arr;
```

Then, you would make an RPC call such as this:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
        xdr_varintarr, &arr...);
```

You could then define `xdr_varintarr` as shown:

```
xdr_varintarr(xdrsp, arrp)
        XDR *xdrsp;
        struct varintarr *arrp;
{
        return (xdr_array(xdrsp, &arrp->data, &arrp->arrlnth,
                MAXLEN, sizeof(int), xdr_int));
}
```

This routine takes 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, and an XDR routine for handling each array element.

If you know the size of the array in advance, you can use `xdr_vector`, which serializes fixed-length arrays, as shown in the following example:

```
int intarr[SIZE];

xdr_intarr(xdrsp, intarr)
        XDR *xdrsp;
        int intarr[];
{
        return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int),
                xdr_int));
}
```

3.1.1.5 XDR Serializing Defaults

XDR always converts quantities to 4-byte multiples when serializing. If the examples in Section 3.1.1.4 had used characters instead of integers, each character would occupy 32 bits. This is why XDR has the built-in routine `xdr_bytes`, which is like `xdr_array` except that it packs characters. The `xdr_bytes` routine has four parameters, similar to the first four of `xdr_array`. For null-terminated strings, XDR provides the built-in routine `xdr_string`, which is the same as `xdr_bytes` without the length parameter.

When serializing, XDR gets the string length from `strlen`, and on deserializing it creates a null-terminated string. The following example calls the user-defined routine `xdr_simple`, as well as the built-in functions `xdr_string` and `xdr_reference` (which locates pointers):

```
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 that `xdr_simple` could be called here instead of `xdr_reference`.

3.1.2 Lowest Layer of RPC

Examples in previous sections show how RPC handles many details automatically through defaults. The following sections describe how to change the defaults by using the lowest layer of the RPC library.

The following capabilities are available only with the lowest layer of RPC:

- It enables you to use TCP as the underlying transport instead of UDP, which restricts RPC calls to only 8K bytes of data.
- It enables you to allocate and free memory explicitly while serializing or deserializing with XDR routines. For more explanation, see Section 3.1.2.3.
- It enables authentication on either the client or server side, through credential verification.

3.1.2.1 The Server Side and the Lowest RPC Layer

The server for the `nusers` program in Example 3–4 does the same work as the previous `nuser` program that used `registerrpc` (see Example 3–3). However, it uses the lowest layer of RPC.

Example 3–4: Server Program Using Lowest 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");
```

Example 3–4: Server Program Using Lowest Layer of RPC (cont.)

```
                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;
        }
}
```

In this example, the server gets a transport handle for receiving and replying to RPC messages. 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 `clntudp_create` (the low-level client routine) must match. The `registerrpc` routine uses `svcudp_create` to get a UDP handle. If you need a more reliable protocol, call `svctcp_create` instead.

After creating a handle with the command `SVCXPRT *transp`, 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 tables.

Finally, associate the program number RUSERSPROG and the version RUSERSVERS with the procedure nuser, which in this case is IPPROTO_UDP. Unlike registerrpc, there are no XDR routines in the registration process, and registration is at the program level rather than the procedure level.

A service can register its port number with the local portmapper service by specifying a non-zero protocol number in the final argument of svc_register. A client determines the server's port number by consulting the portmapper on its server machine. Specifying a zero port number in clntudp_create or clnttcp_create does this automatically.

The user routine nuser must call and dispatch the appropriate XDR routines based on the procedure number. The nusers routine explicitly handles two cases that are taken care of automatically by registerrpc:

- The procedure NULLPROC (currently zero) returns with no 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 nusers 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. It is not necessary to have nusers declared as static here because svc_sendreply is called within that function itself.

To show how a server handles an RPC program that receives data, you could add to the previous example, a procedure called RUSERSPROC_BOOL, which has an argument nusers, and returns TRUE or FALSE depending on whether the number of users logged on is equal to nusers. It would look like this:

```
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;
}
```

Here, the `svc_getargs` routine takes as arguments an `SVCXPRT` handle, the XDR routine, and a pointer to where the input is to be placed.

3.1.2.2 The Client Side and the Lowest RPC Layer

When you use `callrpc`, you cannot control the RPC delivery mechanism and socket that transport the data. The lowest layer of RPC enables you to modify these parameters, as shown in Example 3–5, which calls the `nusers` service.

Example 3–5: Using Lowest RPC Layer to Control Data Transport and Delivery

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.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);
        }
        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);
```

Example 3–5: Using Lowest RPC Layer to Control Data Transport and Delivery (cont.)

```
        }
        printf("%d users on %s\n", nusers, argv[1]);
        clnt_destroy(client);
        exit(0);
}
```

In this example, `CLIENT` pointer is encoded with the transport mechanism. The `callrpc` routine uses UDP and calls `clntudp_create` to get a `CLIENT` pointer; to get TCP you would 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. When the `sin_port` is 0, the remote portmapper is queried to find out the address of the remote service.

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. If the client does not hear from the server within the time specified in `pertry_timeout`, the request may be sent again to the server. The number of tries that `clnt_call` makes to contact the server is equal to the `clnt_call` timeout divided by the `clntudp_create` timeout.

The `clnt_destroy` call always deallocates the space associated with the `CLIENT` handle. It closes the socket associated with the `CLIENT` handle only if the RPC library opened it. If the socket was opened by the user, it remains 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 `clnttcp_create`:

```
clnttcp_create(&server_addr, prognum, versnum, &sock,
               inbufsize, outbufsize);
```

Here, 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, as here, 0 is specified for either of these, the system chooses default values.

The simplest routine that creates a client handle is clnt_create:

```
clnt = clnt_create(server_host, prognum, versnum, transport);
```

The parameters here are the name of the host on which the service resides, the program and version number, and the transport to be used. The transport can be either udp for UDP or tcp for TCP. You can change the default timeouts by using clnt_control. For more information, refer to Section 2.6.

3.1.2.3 Memory Allocation with XDR

To enable memory allocation, 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. For example, the following XDR routine xdr_chararr1, handles 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));
}
```

Here, 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 in this way:

```
xdr_chararr2(xdrsp, chararrp)
        XDR *xdrsp;
        char **chararrp;
{
        int len;

        len = SIZE;
        return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

The RPC call might look like this:

```
char *arrptr;

arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * Use the result here
 */
svc_freeargs(transp, xdr_chararr2, &arrptr);
```

After using the character array, you can free it with `svc_freeargs`; this will not free any memory if the variable indicating it is NULL. For example, in the earlier routine `xdr_finalexample`, if `finalp->string` was NULL, it would not be freed. The same is true for `finalp->simplep`.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory as follows:

- When an XDR routine is called from `callrpc`, the serializing part is used.
- When called from `svc_getargs`, the deserializer is used.
- When called from `svc_freeargs`, the memory deallocator is used.

When building simple examples as shown in this section, you can ignore the three modes. See Appendix A for examples of more sophisticated XDR routines that determine mode and any required modification.

3.2 Raw RPC

Raw RPC refers to the use of pseudo-RPC interface routines that do not use any real transport at all. These routines, `clntraw_create` and `svcraw_create`, help in debugging and testing the non-communications oriented aspects of an application before running it over a real network. Example 3–6 shows their use.

Example 3–6: Debugging and Testing Noncommunication Parts of an Application

```
/*
 * A simple program to increment the number by 1
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpc/raw.h>        /* required for raw */

struct timeval TIMEOUT = {0, 0};
static void server();
```

Example 3–6: Debugging and Testing Noncommunication Parts of an Application (cont.)

```
main()
{
        CLIENT *clnt;
        SVCXPRT *svc;
        int num = 0, ans;

        if (argc == 2)
                num = atoi(argv[1]);
        svc = svcraw_create();
        if (svc == NULL) {
                fprintf(stderr, "Could not create server handle\n");
                exit(1);
        }
        svc_register(svc, 200000, 1, server, 0);
        clnt = clntraw_create(200000, 1);
        if (clnt == NULL) {
                clnt_pcreateerror("raw");
                exit(1);
        }
        if (clnt_call(clnt, 1, xdr_int, &num, xdr_int, &num1,
          TIMEOUT) != RPC_SUCCESS) {
                clnt_perror(clnt, "raw");
                exit(1);
        }
        printf("Client: number returned %d\n", num1);
        exit(0) ;
}

static void
server(rqstp, transp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        int num;

        switch(rqstp->rq_proc) {
        case 0:
                if (svc_sendreply(transp, xdr_void, 0) == NULL) {
                        fprintf(stderr, "error in null proc\n");
                        exit(1);
                }
                return;
        case 1:
                break;
        default:
                svcerr_noproc(transp);
                return;
        }
        if (!svc_getargs(transp, xdr_int, &num)) {
                svcerr_decode(transp);
                return;
        }
        num++;
        if (svc_sendreply(transp, xdr_int, &num) == NULL) {
                fprintf(stderr, "error in sending answer\n");
                exit(1);
        }
        return;
```

Example 3–6: Debugging and Testing Noncommunication Parts of an Application (cont.)

```
}
```

In this example,

- All the RPC calls occur within the same thread of control.
- `svc_run` is not called.
- It is necessary that the server be created before the client.
- `svcraw_create` takes no parameters.
- The last parameter to `svc_register` is 0, which means that it will not register with `portmapper`.
- The server dispatch routine is the same as it is for normal RPC servers.

3.3 Miscellaneous RPC Features

The following sections describe other useful features for RPC programming.

3.3.1 Using Select on the Server Side

Suppose a process simultaneously responds to RPC requests and performs another activity. If the other activity periodically updates a data structure, the process can set an alarm signal before calling `svc_run`. However, if the other activity must wait on a file descriptor, the `svc_run` call does not work. The code for `svc_run` is as follows:

```
void
svc_run()
{
        fd_set readfds;
        int dtbsz = getdtablesize();

        for (;;) {
                readfds = svc_fds;
                switch (select(dtbsz, &readfds, NULL,NULL,NULL)) {

                case -1:
                        if (errno != EBADF)
                                continue;
                        perror("select");
                        return;
                case 0:
                        continue;
                default:
                        svc_getreqset(&readfds);
                }
        }
}
```

You can bypass `svc_run` and call `svc_getreqset` if you know the file descriptors of the sockets associated with the programs you are waiting on. In this way, you can have your own `select` that waits on the RPC socket, and you can have your own descriptors. Note that `svc_fds` is a bit mask of all the file descriptors that RPC uses for services. It can change whenever any RPC library routine is called, because descriptors are constantly being opened and closed; for example, for TCP connections.

Note

If you are handling signals in your application, do not make any system call that accidentally sets `errno`. If this happens, reset `errno` to its previous value before returning from your signal handler.

3.3.2 Broadcast RPC

The portmapper required by broadcast RPC is a daemon that converts RPC program numbers into DARPA protocol port numbers. The main differences between broadcast RPC and normal RPC are the following:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more from each responding server).
- Broadcast RPC is supported only by packet-oriented (connectionless) transport protocols like UDP/IP.
- Broadcast RPC filters out all unsuccessful responses; if a version mismatch exists 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 broadcast RPC.
- Broadcast requests are limited in size to 1400 bytes. Replies can be up to 8800 bytes (the current maximum UDP packet size).

In the following example, the procedure `eachresult` is called each time a response is obtained. It returns a Boolean that indicates whether or not the user wants more responses. If the argument to `eachresult` is NULL, `clnt_broadcast` returns without waiting for any replies:

```
#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 gotten */
```

In the following example, if `done` is TRUE, 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 in a default total timeout period, the routine returns with `RPC_TIMEDOUT`:

```
bool_t done;
  .
  .
  .
done = eachresult(resultsp, raddr)
        caddr_t resultsp;
        struct sockaddr_in *raddr; /* Addr of responding server */
```

For more information, refer to Section 2.7.1.

3.3.3 Batching

In normal RPC, clients send a call message and wait for the server to reply by indicating that the call succeeded. This implies that clients must wait idle while servers process a call. This is inefficient if the client does not want or need an acknowledgment for every message sent. However, calls made by clients are buffered, causing no processing on the servers. When the connection is flushed, a normal RPC request is sent, and processed by the server, which sends back the reply.

RPC messages can be placed in a “pipeline” of calls to a desired server; this is called **batching**, in which:

- No RPC call in the pipeline requires a 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.

Because the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Also, the TCP/IP implementation holds several call messages in a buffer and sends 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. Because the batched calls are buffered, the client must eventually do a nonbatched call to flush the pipeline.

In the following example of batching, assume that a string-rendering service (for example, a window system) has two similar calls: one provides a string and returns void results, and the other provides a string and does nothing else. The service (using the TCP/IP transport) may look like Example 3–7.

Example 3–7: Batching RPC Messages

```
#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");
                        /*
                         * Tell caller he erred
                         */
                        svcerr_decode(transp);
                        return;
                }
                /*
                 * 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");
```

Example 3–7: Batching RPC Messages (cont.)

```
                        /*
                         * 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);
}
```

In this example, the service could have one procedure that takes the string and a Boolean to indicate whether or not the procedure will respond. For a client to use batching effectively, the client must perform RPC calls on a TCP-based transport and the actual calls must have the following attributes:

- The XDR routine of the result must be zero (`NULL`).
- The timeout of the RPC call must be zero. (Do not rely on `clnt_control` to assist in batching.)

If a UDP transport is used instead, the client call becomes a message to the server and the RPC mechanism becomes simply a message passing system, with no batching possible. In Example 3–8, a client uses batching to supply several strings; batching is flushed when the client gets a null string (EOF).

Example 3–8: Client Batching

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <suntool/windows.h>

main(argc, argv)
        int argc;
        char **argv;
{
        struct timeval total_timeout;
        register CLIENT *client;
        enum clnt_stat clnt_stat;
        char buf[1000], *s = buf;

        if ((client = clnt_create(argv[1],
          WINDOWPROG, WINDOWVERS, "tcp")) == NULL) {
                perror("clnttcp_create");
                exit(-1);
        }
        total_timeout.tv_sec = 0;       /* set timeout to zero */
```

Example 3–8: Client Batching (cont.)

```
        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, "batching 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, "batching rpc");
                exit(-1);
        }
        clnt_destroy(client);
        exit(0);
}
```

In this example, the server sends no message, making the clients unable to receive notice of any failures that may occur. Therefore, the clients must handle any errors. This example was completed to render all of the lines (approximately 2000) in the file `/etc/termcap`. The rendering service simply discarded the entire file. The example was run in four configurations, in different amounts of time:

- 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 only `fscanf` on `/etc/termcap` requires 6 seconds. These timings show the advantage of protocols that enable overlapped execution, although they are difficult to design.

3.3.4 Authentication of RPC Calls

In the examples presented so far, the caller never identified itself to the server, nor did the server require it from the caller. 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 `none`.

The authentication subsystem of the RPC package, with its ability to create and send authentication parameters, can support commercially available authentication software. This manual describes only one type of authentication, authentication through the operating system.

3.3.5 Authentication Through the Operating System

The following sections describe client and server side authentication through the operating system.

3.3.5.1 The Client Side

Assume that a caller creates the following new RPC client handle:

```
clnt = clntudp_create(address, prognum, versnum, wait, sockp)
```

The transport for this client handle defaults to the following associate authentication handle:

```
clnt->cl_auth = authnone_create();
```

The RPC client can choose to use authentication that is native to the operating system 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:

```
/*
 * credentials native to the operating system
 */
struct authunix_parms {
        u_long   aup_time;          /* credentials creation time */
        char     *aup_machname;     /* host name where client is */
        int      aup_uid;           /* client’s UNIX 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 */
};
```

In this example, the fields are set by `authunix_create_default` by invoking the appropriate system calls. Because the RPC user created this new style of authentication, the user is responsible for destroying it (to save memory) with the following:

```
auth_destroy(clnt->cl_auth);
```

3.3.5.2 The Server Side

It is difficult for service implementors to handle authentication because 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 vers num */
        u_long   rq_proc;           /* desired procedure number */
        struct opaque_auth rq_cred; /* raw credentials from wire */
        caddr_t  rq_clntcred;       /* credentials (read only) */
};
```

The `rq_cred` is mostly opaque, except for one field, the style 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:

- The `rq_cred` field of the request is well-formed; that is, the service implementor can use the `rq_cred.oa_flavor` field of the request to determine the authentication style used by the caller. The service implementor can also inspect other fields of `rq_cred` if the style is not supported by the RPC package.
- The `rq_clntcred` field of the request is either NULL or points to a well-formed structure that corresponds to a supported style of authentication credentials.

The `rq_clntcred` field also could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is NULL, the service implementor can inspect the other (opaque) fields of `rq_cred` to determine whether the service knows about a new type of authentication that is unknown to the RPC package.

Example 3–9 extends the previous remote users service (see Example 3–3) so that it computes results for all users except UID 16.

Example 3–9: Modifying the Remote Users Service

```
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
```

Example 3–9: Modifying the Remote Users Service (cont.)

```
         */
        if (rqstp->rq_proc == NULLPROC) {
                if (!svc_sendreply(transp, xdr_void, 0))
                        fprintf(stderr, "can’t reply to RPC call\n");
                 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:        /* return weak authentication error */
                svcerr_weakauth(transp);
                return;
        }
        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;

        default:
                svcerr_noproc(transp);
                return;
        }
}
```

As in this example, it is not customary to check the authentication parameters associated with `NULLPROC` (procedure number zero). Also, if the authentication parameter type is not suitable for your service, have your program call `svcerr_weakauth`.

The service protocol itself returns status for access denied; in this example, the protocol does not do this; instead, a call to the service primitive, `svcerr_systemerr`, is made. RPC deals only with authentication and not with the access control of an individual service. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

3.3.6 Using the Internet Service Daemon (inetd)

You can start an RPC server from `inetd`. The only difference from the usual code is that it is best to have the service creation routine called in the following form because `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, call `svc_register` as follows, with the last parameter flag set to `0`, because the program is already registered with the portmapper by `inetd`:

```
svc_register(transp, PROGNUM, VERSNUM, service, 0);
```

If you want to exit from the server process and return control to `inetd`, you must do so explicitly, because `svc_run` never returns.

The format of entries in `/etc/inetd.conf` for RPC services is in one of the following two forms:

```
p_name/version dgram  rpc/udp wait/nowait user server args

p_name/version stream rpc/tcp wait/nowait user server args
```

The variable, *`p_name`*, is the symbolic name of the program as it appears in the file `/etc/rpc`; *`server`* is the program implementing the server; and *`program`* and *`version`* are the program and version numbers of the service. For more information, see `inetd.conf`(5)

If the same program handles multiple versions, then the version number can be a range, as in this example:

```
rstatd/1-2 dgram rpc/udp wait root /usr/etc/rpc.rstatd
```

3.4 Additional Examples

The following sections present additional examples for server and client sides, TCP, and callback procedures.

3.4.1 Program Versions on the Server Side

By convention, the first version of program `PROG` is designated as `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 you name this version `RUSERSVERS_SHORT`, then a server that wants to support both versions would register both. It is not necessary to create another server handle for the new version, as shown in this segment of code:

```
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 new service\n");
        exit(1);
}
```

You can handle both versions with the same C procedure, as in Example 3–10.

Example 3–10: C Procedure that Returns Two Different Data Types

```
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;
        }
        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;
```

Example 3–10: C Procedure that Returns Two Different Data Types (cont.)

```
        }
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

3.4.2 Program Versions on the Client Side

The network can have different versions of an RPC server. For example, one server might run `RUSERSVERS_ORIG`, and another might run `RUSERSVERS_SHORT`.

If the version of the server running does not match the version number in the client creation routines, then `clnt_call` fails with a `RPCPROGVERSMISMATCH` error. You can determine the version numbers supported by the server and then create a client handle with an appropriate version number. To do this, use `clnt_create_vers` (refer to `rpc`(3) for more information) or the routine shown in Example 3–11.

Example 3–11: Determining Server-Supported Versions and Creating Associated Client Handles

```
main()
{
        enum clnt_stat status;
        u_short num_s;
        u_int num_l;
        struct rpc_err rpcerr;
        int maxvers, minvers;

        clnt = clnt_create(host, RUSERSPROG,
                RUSERSVERS_SHORT, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror("clnt");
                exit(-1);
        }
        to.tv_sec = 10;        /* set the time outs */
        to.tv_usec = 0;
        status = clnt_call(clnt, RUSERSPROC_NUM,
                xdr_void, NULL, xdr_u_short, &num_s, to);
        if (status == RPC_SUCCESS) {   /* We found the latest version number */
                clnt_destroy(clnt);
                printf("num = %d\n",num_s);
                exit(0);
        }
        if (status != RPC_PROGVERSMISMATCH) {  /* Some other error */
                clnt_perror(clnt, "rusers");
                exit(-1);
        }
        clnt_geterr(clnt, &rpcerr);
        maxvers = rpcerr.re_vers.high;  /* highest version supported */
```

Example 3–11: Determining Server-Supported Versions and Creating Associated Client Handles (cont.)

```
        minvers = rpcerr.re_vers.low;   /* lowest version supported */
        if (RUSERSVERS_ORIG < minvers ||
                RUSERS_ORIG > maxvers) {
                /* doesn't meet minimum standards */
                clnt_perror(clnt, "version mismatch");
                exit(-1);
        }
        /* This version not supported */
        clnt_destroy(clnt);          /* destroy the earlier handle */
        clnt = clnt_create(host, RUSERSPROG,
                RUSERSVERS_ORIG, "udp");   /* try different version */
        if (clnt == NULL) {
                clnt_pcreateerror("clnt");
                exit(-1);
        }
        status = clnt_call(clnt, RUSERSPROCNUM,
                xdr_void, NULL, xdr_u_long, &num_l, to);
        if (status == RPC_SUCCESS) {  /* We found the latest version number */
                printf("num = %d\n", num_l);
        } else {
                clnt_perror(clnt, "rusers");
                exit(-1);
        }
}
```

3.4.3 Using the TCP Transport

Example 3–12 works like the remote file copy command `rcp`; that is, the initiator of the RPC call, `snd`, takes its standard input and sends it to the server `rcv`, which prints it on standard output; the RPC call uses TCP. The example also shows how an XDR procedure behaves differently on serialization than on deserialization.

Example 3–12: RPC Call That Uses TCP Protocol

```
/*
 * 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_rcp(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) {
```

Example 3–12: RPC Call That Uses TCP Protocol (cont.)

```
                        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);
                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 "rcp.h"        /* for prog, vers definitions */

main(argc, argv)
        int argc;
        char **argv;
{
        int xdr_rcp();
        int err;

        if (argc < 2) {
                fprintf(stderr, "usage: %s servername\n", argv[0]);
                exit(-1);
        }
        if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC,
          RCPVERS, xdr_rcp, 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;
{
        struct sockaddr_in server_addr;
        int socket = RPC_ANYSOCK;
        enum clnt_stat clnt_stat;
        struct hostent *hp;
        register CLIENT *client;
        struct timeval total_timeout;
```

Example 3–12: RPC Call That Uses TCP Protocol (cont.)

```
        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,
                hp->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) {
                clnt_createerror("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>
#include "rcp.h"        /* for prog, vers definitions */

main()
{
        register SVCXPRT *transp;
        int rcp_service(), xdr_rcp();

        if ((transp = svctcp_create(RPC_ANYSOCK,
                BUFSIZ, BUFSIZ)) == NULL) {
                fprintf("svctcp_create: error\n");
                exit(1);
        }
        pmap_unset(RCPPROG, RCPVERS);
        if (!svc_register(transp, RCPPROG,
                RCPVERS, rcp_service, IPPROTO_TCP)) {
                fprintf(stderr, "svc_register: error\n");
                exit(1);
        }
        svc_run();  /* never returns */
        fprintf(stderr, "svc_run should never return\n");
}
rcp_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: rcp_service");
                return;
        case RCPPROC_FP:
                if (!svc_getargs(transp, xdr_rcp, stdout)) {
                        svcerr_decode(transp);
                        return;
                }
                if (!svc_sendreply(transp, xdr_void, 0))
```

Example 3–12: RPC Call That Uses TCP Protocol (cont.)

```
                        fprintf(stderr, "can’t reply\n");
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

3.4.4 Callback Procedures

It is sometimes useful to have a server become a client, and make an RPC call back to the process that is its client. An example of this is remote debugging, where the client is a window-system program and the server is a debugger running on the remote machine. Mostly, the user clicks a mouse button at the debugging window (converting 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 reaches a breakpoint, the roles are reversed, and the debugger wants to make an RPC call to the window program, so that it can tell the user that a breakpoint has been reached.

Callbacks are also useful when the client cannot block (that is, wait) to hear back from the server (possibly because of excessive processing in serving the request). In such cases, the server could acknowledge the request and use a callback to reply.

To do an RPC callback, you need a program number on which to make the RPC call. The program number is dynamically generated, so it must be in the transient range `0x40000000`. The routine `gettransient` returns a valid program number in the transient range, and registers it with the portmapper. It only communicates with the portmapper running on the same machine as the `gettransient` routine itself.

The call to `pmap_set` is a test-and-set operation, because it indivisibly tests whether or not a program number has been registered; if not, it is reserved. The following example shows the `gettransient` routine:

```
#include <stdio.h>
#include <rpc/rpc.h>

gettransient(proto, vers, portnum)
        int proto;
        u_long vers;
        u_short portnum;
{
        static u_long prognum = 0x40000000;
        while (!pmap_set(prognum++, vers, proto, portnum))
                continue;
        return (prognum - 1);
}
```

Note that the call to `ntohs` for `portnum` is unnecessary because it was already passed in host byte order (as `pmap_set` expects).

The following list describes how the client/server programs in Example 3–13 use the `gettransient` routine:

- The client makes an RPC call to the server, passing it a transient program number.
- The client 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.

In Example 3–13, both the client and the server are on the same machine; otherwise, host name handling would be different.

Example 3–13: Client-Server Usage of gettransient Routine

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include "example.h"

int callback();

main()
{
        int tmp_prog;
        char hostname[256];
        SVCXPRT *xprt;
        int stat;

        gethostname(hostname, sizeof(hostname));
        if ((xprt = svcudp_create(RPC_ANYSOCK)) == NULL) {
                fprintf(stderr, "rpc_server: svcudp_create\n");
                exit(1);
        }
        if (tmp_prog = gettransient(IPPROTO_UDP, 1,
                        xprt->xp_port) == 0) {
                fprintf(stderr, "failed to get transient number\n");
                exit(1);
        }
        fprintf(stderr, "client gets prognum %d\n", tmp_prog);

        /* protocol is 0 - gettransient does registering */
```

Example 3–13: Client-Server Usage of gettransient Routine (cont.)

```
        (void)svc_register(xprt, tmp_prog, 1, callback, 0);
        stat = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
          EXAMPLEPROC_CALLBACK, xdr_int, &tmp_prog, xdr_void, 0);

        if (stat != RPC_SUCCESS) {
                clnt_perrno(stat);
                exit(1);
        }
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn’t return\n");
}

callback(rqstp, transp)
        register struct svc_req *rqstp;
        register SVCXPRT *transp;
{
        switch (rqstp->rq_proc) {

                case 0:
                        if (!svc_sendreply(transp, xdr_void, 0)) {
                                fprintf(stderr, "err: exampleprog\n");
                                return (1);
                        }
                        return (0);

                case 1:
                        fprintf(stderr, "client got callback\n");
                        if (!svc_sendreply(transp, xdr_void, 0)) {
                                fprintf(stderr, "err: exampleprog\n");
                                return (1);
                        }
        }
        return (0);
}
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>
#include "example.h"

char *getnewprog();
char hostname[256];
int docallback();
int pnum = -1;                /* program number for callback routine */

main()
{
        gethostname(hostname, sizeof(hostname));
        registerrpc(EXAMPLEPROG, EXAMPLEVERS,
          EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
        signal(SIGALRM, docallback);
        alarm(10);
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn’t return\n");
}

char *
getnewprog(pnump)
        int *pnump;
```

Example 3–13: Client-Server Usage of gettransient Routine (cont.)

```
{
        pnum = *(int *)pnump;
        return NULL;
}

docallback()
{
        int ans;

        if (pnum == -1) {
                signal(SIGALRM, docallback);
                return;     /* program number not yet received */
        }
        ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
                xdr_void, 0);
        if (ans != RPC_SUCCESS)
                fprintf(stderr, "server: %s\n",clnt_sperrno(ans));
}
```

A
External Data Representation: Technical Notes

This appendix contains technical notes on the External Data Representation (XDR) standard, a set of library routines that enable C programmers to describe arbitrary data structures in a machine-independent way. For a formal specification of the XDR standard, see RFC1014 , *External Data Representation: Protocol Specification*.

XDR is the backbone of the Open Network Computing Remote Procedure Call (ONC RPC) package, because data for remote procedure calls is transmitted using the XDR standard. Use the XDR library routines to transmit data that is read or written from several types of machine. For a complete specification of the system External Data Representation routines, see the `xdr`(3) reference page.

This appendix also 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) only need the information on number filters (Section A.1.3.1), floating-point filters (Section A.1.3.2), and enumeration filters (Section A.1.3.3). Programmers who want to implement RPC and XDR on new machines should read the rest of the appendix.

Note

You can use `rpcgen` to write XDR routines regardless of whether RPC calls are being made.

C programs that need XDR routines must include the file `<rpc/rpc.h>`, which contains all necessary interfaces to the XDR system. The C library `libc.a` contains all the XDR routines, so you can compile as usual.

A.1 Usefulness of XDR

Consider the following two programs, `writer.c` and `reader.c`:

```
#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);
}

#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);
}
```

The two programs appear to be portable because:

- They pass `lint` checking.
- They work the same when executed on two different hardware architectures, a Sun and a MIPS.

Piping the output of the `writer.c` program to the `reader.c` program gives identical results on a MIPS or a Sun, as shown:

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%

mips% writer | reader
0 1 2 3 4 5 6 7
mips%
```

With local area networks and Berkeley UNIX 4.2BSD came the concept of network pipes, in which a process produces data on one machine, and a second process on another machine uses this data. You can construct a network pipe with `writer.c` and `reader.c`. Here, the first process (on a Sun) produces data used by a second process (on a MIPS):

```
sun% writer | rsh mips reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
sun%
```

You get identical results by executing `writer.c` on the MIPS and `reader.c` on the Sun. These results occur because the byte ordering of long integers differs between the MIPS and the Sun, although the word size is the same. Note that `16777216` is equal to 2^{24}; when four bytes are reversed, the `1` is in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. You can make programs data-portable by replacing the `read` and `write` calls with calls to an XDR library routine `xdr_long`, which is a filter that recognizes the standard representation of a long integer in its external form. Here are the revised versions of `writer.c` (Example A–1) and `reader.c` (Example A–2):

Example A–1: Revised Version of writer.c

```
#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);
}
```

Example A–2: Revised Version of reader.c

```
#include <stdio.h>
#include <rpc/rpc.h> /* XDR is a sub-library of RPC */

main()  /* reader.c */
{
```

Example A–2: Revised Version of reader.c (cont.)

```
 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 were executed on a MIPS, a Sun, and from a Sun to a MIPS; the results are as follows:

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%

mips% writer | reader
0 1 2 3 4 5 6 7
mips%

sun% writer | rsh mips reader
0 1 2 3 4 5 6 7
sun%
```

Note

Arbitrary data structures create portability problems, particularly with alignment and pointers:

- Alignment on word boundaries may cause the size of a structure to vary on different machines.
- A pointer has no meaning outside the machine where it is defined.

A.1.1 A Canonical Standard

The XDR approach to standardizing data representations is **canonical**, because XDR defines a single byte order (big-endian), a single floating-point representation (IEEE), and so on. A program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard. Similarly, any such program can read portable data by translating the XDR standard representation to the local equivalent.

The single standard treats separately those programs that create or send portable data and those that use or receive the data. A new machine or language has no effect upon existing portable data creators and users. Any new machine simply uses the canonical standards of XDR; the local representations of other machines are irrelevant. To existing programs on other machines, the local representations of the new machine are also irrelevant. There are strong precedents for the canonical approach of XDR. For example, TCP/IP, UDP/IP, XNS, Ethernet, and 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.

The canonical approach does have one disadvantage of little practical importance. Suppose two little-endian machines transfer integers according to the XDR standard. The sending machine converts the integers from little-endian byte order to XDR (big-endian) byte order, and the receiving machine does the reverse. Because both machines observe the same byte order, the conversions were really unnecessary. Fortunately, 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.

A.1.2 The XDR Library

The XDR library enables you to write and read arbitrary C constructs consistently. This makes it useful even when the data is not shared among machines on a network. The XDR library can do this because it has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays. 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.

The previous `writer.c` and `reader.c` routines manipulate data by using standard I/O routines, so `xdrstdio_create` was used. The parameters to XDR stream creation routines vary according to their function. For example, `xdrstdio_create` takes the following parameters:

- A pointer to an XDR structure that it initializes
- A pointer to a `FILE` that the input or output acts upon

- The operation — either XDR_ENCODE for serializing in writer.c or XDR_DECODE for deserializing in reader.c

It is not necessary for RPC users to create XDR streams; the RPC system itself can create these streams and pass them to the users. There is a family of XDR stream creation routines in which each member treats the stream of bits differently.

The xdr_long primitive is characteristic of most XDR library primitives and all client XDR routines for two reasons:

- The routine returns FALSE (0) if it fails, and TRUE (1) if it succeeds.
- For each data type xxx, there is an associated XDR routine of the form:

  ```
  xdr_xxx(xdrs, xp)
   XDR *xdrs;
   xxx *xp;
  {
  }
  ```

In this case, xxx is long, and the corresponding XDR routine is a primitive, xdr_long. The client could also define an arbitrary structure xxx in which case the client would also supply the routine xdr_xxx, describing each field by calling XDR routines of the appropriate type. In all cases, the first parameter, xdrs, is 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 important for portable data. Calling the same routine for either operation practically guarantees that serialized data can also be deserialized. Thus, one routine is used by both producer and consumer of networked data.

You implement direction independence by passing the address of an object rather than the object itself (only with deserialization is the object modified). If needed, the user can obtain the direction of the XDR operation. See Section A.1.5 for details.

For a more complicated example, assume that a person's gross assets and liabilities are to be exchanged among processes, and each is a separate data type:

```
struct gnumbers {
 long g_assets;
 long g_liabilities;
};
```

The corresponding XDR routine describing this structure would be 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);
}
```

In this example, the parameter, `xdrs`, is never inspected or modified; it is only passed to subcomponent routines. The program must inspect the return value of each XDR routine call and stop immediately and return FALSE upon subroutine failure.

This example also shows that the type `bool_t` is declared as an integer whose only value is TRUE (`1`) or FALSE (`0`). The following definitions apply:

```
#define bool_t int
#define TRUE 1
#define FALSE 0
```

With these conventions, you can rewrite `xdr_gnumbers` as follows:

```
xdr_gnumbers(xdrs, gp)
 XDR *xdrs;
 struct gnumbers *gp;
{
 return(xdr_long(xdrs, &gp->g_assets) &&
  xdr_long(xdrs, &gp->g_liabilities));
}
```

Either coding style can be used.

A.1.3 XDR Library Primitives

The following sections describe the XDR primitives (basic and constructed data types) and XDR utilities. The include file `<rpc/xdr.h>`, (automatically included by `<rpc/rpc.h>`) defines the interface to these primitives and utilities.

A.1.3.1 Number Filters

The XDR library provides primitives that translate between numbers and their corresponding external representations. Primitives include 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_hyper(xdrs, hp)
 XDR *xdrs;
 longlong_t *hp;

bool_t xdr_u_hyper(xdrs, uhp)
 XDR *xdrs;
 u_longlong_t *uhp;

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_longlong_t(xdrs, hp)
 XDR *xdrs;
 longlong_t *hp;

bool_t xdr_u_longlong_t(xdrs, uhp)
 XDR *xdrs;
 u_long *uhp;

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 otherwise.

For more information on number filters, see the `xdr`(3) reference page.

A.1.3.2 Floating Point Filters

The XDR library also provides primitive routines for floating point types in C:

```
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 otherwise.

Note

Because 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.

A.1.3.3 Enumeration Filters

The XDR library provides a primitive for generic enumerations; it assumes that a C `enum` has the same representation inside the machine as a C `integer.` The Boolean type is an important instance of the `enum`. The external representation of a Boolean is always TRUE (`1`) or FALSE (`0`) as shown here:

```
#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`.

A.1.3.4 Possibility of No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The following routine does this:

```
bool_t xdr_void();  /* always returns TRUE */
```

A.1.3.5 Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives previously discussed. The following sections include primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with XDR_DECODE. XDR enables memory deallocation through the XDR_FREE operation. The three XDR directional operations are XDR_ENCODE, XDR_DECODE, and XDR_FREE.

A.1.3.5.1 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. When a string is passed or manipulated, there must be a pointer to it. Therefore, the XDR library defines a string to be a `char *`, not a sequence of characters. The external and internal representations of a string are different. Externally, strings are represented as sequences of ASCII characters; internally, with character pointers. The `xdr_string` routine converts between the two, as shown:

```
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, `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 may specify that a file name cannot be longer than 255 characters. Keep `maxlength` small because overflow conditions may occur if `xdr_string` has to call `malloc` for space. The routine returns FALSE if the number of characters exceeds `maxlength`; otherwise, it returns TRUE.

The behavior of `xdr_string` is similar to that of other routines in this section. For the direction, `XDR_ENCODE`, 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 XDR assumes that a target area (which can hold strings no longer than `maxlength`) has been allocated. In either case, the string is decoded into the target area, and the routine appends a NULL character to it.

In the `XDR_FREE` operation, the string is obtained by dereferencing `sp`. If the string is not NULL, it is freed and `*sp` is set to NULL. In this operation, `xdr_string` ignores the `maxlength` parameter.

A.1.3.5.2 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 and internal byte representation is the same.

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 are identical to the same parameters of `xdr_string`. The length of the byte area is obtained by dereferencing `lp` when serializing; `*lp` is set to the byte length when deserializing.

A.1.3.5.3 Arrays

The XDR library 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` in addition to two more: the size of array elements, and an XDR routine to handle each of the elements.

This routine encodes or decodes each array element:

```
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 allowable number of array elements; `elementsiz` is the byte size of each array element. (You can also use the C function `sizeof` to obtain this value.) The `xdr_element` routine is called to serialize, deserialize, or free each element of the array.

Consider the following three examples, which show the recursiveness of the XDR library routines already discussed.

- A user on a networked machine can be identified in three ways:
  - The machine name, such as `krypton`; (see the `gethostname`(2) reference page)
  - The user's UID; (see the `geteuid`(2) reference page)
  - The group numbers to which the user belongs; (see the `getgroups`(2) reference page)

  A structure with this information and its associated XDR routine could be coded like this:

  ```
  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));
  }
  ```

- A party of network users could be implemented as an array of `netuser` structure. The declaration and its associated XDR routines are as follows:

  ```
  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));
  }
  ```

- The parameters to `main`— `argc` and `argv`— can be combined into a structure, and an array of these structures can make up a history of commands. The declarations and XDR routines might look like:

  ```
  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_wrapstring(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_wrapstring));
  }
  ```

```
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 routine xdr_wrapstring is needed to package the xdr_string routine, because the implementation of xdr_array only passes two parameters to the array element description routine; xdr_wrapstring supplies the third parameter to xdr_string.

A.1.3.5.4 Opaque Data

Some protocols pass handles from a server to a client. The client later passes back the handles, without first inspecting them; that is, handles are opaque. The xdr_opaque primitive describes fixed-size, opaque bytes:

```
bool_t xdr_opaque(xdrs, p, len)
 XDR *xdrs;
 char *p;
 u_int len;
```

The parameter p is the location of the bytes; len is the number of bytes in the opaque object. By definition, the data within the opaque object is not machine-portable.

A.1.3.5.5 Arrays of Fixed Size

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);
}
```

A.1.3.5.6 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 */
```

In this example, the routine translates the discriminant of the union at `*dscmp`. The discriminant is always an `enum_t`. Next, the union 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 `proc` is called to translate the union. The end of the `xdr_discrim` structure array is denoted by a routine of value NULL. If the discriminant is not in the `arms` array, then the `defaultarm` procedure is called if it is non-null; otherwise the routine returns FALSE.

The following example shows how to serialize or deserialize a discriminated union. Suppose that the type of a union is an integer, character pointer (a string), or a `gnumbers` structure. Also, assume the union and its current type are declared in a structure, as follows:

```
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 serialize or deserialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
 { INTEGER, xdr_int },
 { GNUMBERS, xdr_gnumbers }
 { STRING, xdr_wrapstring },
 { __dontcare__, NULL }
 /* always terminate arms with a NULL xdr_proc */
}

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 in Section A.1.2 and `xdr_wrapstring` was presented in Example C in Section A.1.3.5.3. The default `arm` parameter to `xdr_union` (the last parameter) is NULL in Example D. Therefore, the value of the union's discriminant can only be a value listed in the `u_tag_arms` array. Example D also shows that the elements of the arm's array do not need to be sorted.

The values of the discriminant may be sparse, though in Example D they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This will document the external representation of the discriminant and guarantee that different C compilers provide identical discriminant values.

A.1.3.5.7 Pointers

In C it is useful to put within a structure any pointers to another structure. The `xdr_reference` primitive makes it easy to serialize, deserialize, and free these referenced structures. A structure of structure pointers is shown here:

```
bool_t xdr_reference(xdrs, pp, size, proc)
 XDR *xdrs;
 char **pp;
 u_int ssize;
 bool_t (*proc)();
```

Parameter pp is the address of the pointer to the structure, ssize is the size in bytes of the structure (use the C function sizeof to obtain this value), and proc is the XDR routine that describes the structure. When decoding data, storage is allocated if *pp is NULL.

There is no need for a primitive xdr_struct to describe a structure within a structure, because pointers are always sufficient.

Note

The xdr_reference and xdr_array primitives are not interchangeable external representations of data.

The following example describes a structure (and its corresponding XDR routine) that contains an item of data and a pointer to a gnumbers structure that has more information about that item 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. This structure has the following construct:

```
struct pgn {
 char *name;
 struct gnumbers *gnp;
};
```

This structure has the following corresponding XDR routine:

```
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);
}
```

In many applications, C programmers attach double meaning to the values of a pointer. Typically the value NULL means data is not necessary, but 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 example, in the previous structure, 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` cannot 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 serializing data will most likely cause a memory fault and a core dump.

The `xdr_pointer` correctly handles NULL pointers. For more information about its use, see Section A.2.

A.1.4 Non-filter Primitives

The non-filter primitives that follow are for manipulating XDR streams:

```
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.

Note

In some XDR streams, the returned value of `xdr_getpos` is meaningless; the routine returns a –1 in this case (though –1 should be a legitimate value).

The routine `xdr_setpos` sets a stream position to `pos`. However, in some XDR streams, setting a position is impossible; in such cases, `xdr_setpos` returns FALSE. This routine also fails if the requested position is out-of-bounds. The definition of bounds varies according to the stream.

The `xdr_destroy` primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

A.1.5 XDR Operation Directions

Though not recommended, you may want to optimize XDR routines by using the direction of the operation — XDR_ENCODE, XDR_DECODE, or XDR_FREE. For example, the value `xdrs->x_op` contains the direction of the XDR operation. An example in Section A.2 shows the usefulness of the xdrs->x_op field.

A.1.6 XDR Stream Access

An **XDR stream** is obtained by calling the appropriate creation routine, which takes arguments for the specific properties of the stream. Streams currently exist for serialization or deserialization of data to or from standard I/O FILE streams, TCP/IP connections and files, and memory.

A.1.6.1 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.

A.1.6.2 Memory Streams

A **memory stream** enables 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;
 u_int len;
 enum xdr_op x_op;
```

The routine xdrmem_create initializes an XDR stream in local memory that is pointed to by parameter addr; parameter len is the length in bytes of the memory. The parameters xdrs and x_op are identical to the corresponding parameters of xdrstdio_create. Currently, the UDP/IP implementation of ONC RPC uses xdrmem_create. Complete call or result messages are built in memory before calling the sendto system routine.

A.1.6.3 Record (TCP/IP) Streams

A **record stream** is an XDR stream built on top of a record marking standard; that is, in turn, built on top of a file or a Berkeley UNIX 4.2BSD connection interface, as shown:

```
#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 ordinary files.

The parameter `xdrs` is similar to the corresponding parameter described in Section A.1.6.2. 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, defaults are used. When a buffer needs to be filled or flushed, the routine `readproc` or `writeproc` is called, respectively. The usage of these routines is similar to the system calls `read` and `write`. However, the first parameter to each routine 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 enables you to delimit records in the byte stream. This is discussed in Section A.2. The following primitives are specific to record streams:

```
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. This does not mean that there is no more data in the underlying file descriptor.

A.1.7 XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

The following structure defines the 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 is not expected to affect the implementation of a stream. The fields x_private, x_base, and x_handy pertain to a particular stream implementation. The field x_public is for the XDR client and must not be used by the XDR stream implementations or the XDR primitives. The macros x_getpostn, x_setpostn, and x_destroy, access 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. To the stream, 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 maximizing efficient use of machine cycles. The resulting buffer is not data-portable, so using this feature is not recommended.)

The operations x_getbytes and x_putbytes get and put sequences of bytes from or to the underlying stream; they return TRUE if successful, and FALSE otherwise. The routines have identical parameters (replace xxx):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
 XDR *xdrs;
 char *buf;
 u_int bytecount;
```

The x_getlong and x_putlong routines receive and put long numbers to and from the data stream. These routines must translate the numbers between the machine representation and the (standard) external representation. The operating system primitives htonl and ntohl help to do 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 creation routine.

A.2 Advanced Topics

This section describes advanced techniques for passing data structures, such as linked lists (of arbitrary length). The examples in this section are written using both the XDR C library routines and the XDR data description language.

The following example presents a C data structure and its associated XDR routines for an individual's gross assets and liabilities. The example is duplicated here:

```
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);
}
```

If you want to implement a linked list of such information, you could construct the following data structure:

```
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 indicates whether 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;
};
```

Here, 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 TRUE, then it is followed by a gnumbers structure and (recursively) by a gnumbers_list. Note that the C declaration has no Boolean explicitly declared in it (though the gn_next field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it. From the XDR description in the previous paragraph, you can determine how to write the XDR routines for a gnumbers_list. That is, the xdr_pointer primitive would implement the XDR union. Unfortunately — due to recursion — using XDR on a list with the following routines causes the C stack to grow linearly with respect to the number of nodes in the list:

```
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 following routine combines these two mutually recursive routines into a single, non-recursive one:

```
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 Boolean information can be serialized. Notice that this is unnecessary in the XDR_DECODE case, because the value of `more_data` is not known until it is deserialized in the next statement, which uses XDR on the `more_data` field of the XDR union. If there is no more data, this last pointer is set to NULL to indicate the list end, and a TRUE is returned to indicate completion. 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, the value of `nextp` is set to indicate the location of the next pointer in the list. This is for dereferencing `gnp` to find the location of the next item in the list; after the next statement, the storage pointed to by `gnp` is deallocated and is no longer valid. This cannot be done for all directions because, in the XDR_DECODE direction, the value of `gnp` is not set until the next statement.

Next, XDR operates on the data in the node through the primitive `xdr_reference`, which is like `xdr_pointer` (which was used before). However, `xdr_reference` does not send over the Boolean indicating whether there is more data; it is used instead of `xdr_pointer` because XDR has already been used on this information. Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is `xdr_gnumbers`, for using XDR on `gnumbers`; however, each element in the list is of type `gnumbers_node`. The `xdr_gnumbers_node` is not passed because it is recursive; instead, use `xdr_gnumbers`, which uses XDR on all of the non-recursive part. Note that this works only if the `gn_numbers` field is the first item in each element, so that the addresses are identical when passed to `xdr_reference`.

Next, `gnp` is updated to point to the next item in the list. If the direction is XDR_FREE, it is set to the previously saved value; otherwise, `gnp` is dereferenced to get the proper value. Although more difficult to understand than the recursive version, the non-recursive routine is much less likely to overflow the C stack. It also runs more efficiently because a lot of procedure call overhead has been removed. Most lists are small though (in the hundreds of items or less) and the recursive version should be sufficient for them.

Index