CHAPTER 2

Using Remote Procedure Calls

This chapter describes the use of Remote Procedure Calls (RPCs). It includes these sections:

- Overview

  Introduces the concept of Remote Procedure Calls (RPCs) as high-level communications mechanisms.
- Higher Layers of RPC

  Describes the highest and intermediate layers of RPC.
- Lowest Layer of RPC

  Describes the lowest level of RPC programs.
- Useful RPC Features

  Discusses some use of select on the server side, broadcast RPC, batching and authentication.
- Programming Examples

  Provides examples of the use of version numbers, a Unix remote file copy program, and callback procedures.

Overview

This document assumes a working knowledge of network theory. It is intended for programmers who wish to write network applications using Remote Procedure Calls (RPC), and who want to understand the RPC mechanisms usually hidden by the rpcgen protocol compiler rpcgen is described in detail in Chapter 4, Using rpcgen.

Note Before attempting to write a network application, or to convert an existing non-network application to run over the network, you may want to understand the material in this chapter. However, for most applications, you can bypass the material presented here by using rpcgen. The section Generating XDR Routines contains the complete source for a working RPC service—a remote directory listing service that uses rpcgen to generate XDR routines as well as client and server stubs.

Remote Procedure Calls (RPCs) are high-level communications mechanisms. RPC presumes the existence of low-level networking mechanisms (such as TCP/IP and UDP/IP), and implements on them a logical client-to-server communications system designed specifically for the support of network applications.

With RPC, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and returns the procedure call to the client.

Layers of RPC

The RPC interface can be seen as being divided into three layers.

The Highest Layer

The highest layer is totally transparent to the operating system, machine, and network on which it is run. Think of this level as a way of using RPC, rather than as a part of RPC itself.

Programmers who write RPC routines usually make this layer available to others by way of a simple C language front end that entirely hides the networking.

At this level, a program can simply make a call to rnusers(), a C routine that returns the number of users on a remote machine. Users are not explicitly aware of using RPC – they simply call a procedure, just as they would call malloc().

The Middle Layer

The middle layer is really the heart of RPC. Here, the user does not need to consider details about sockets, the UNIX system, or other low-level implementation mechanisms. They simply make remote procedure calls to routines on other machines. The inherent value of this layer is its simplicity. It is this layer that allows RPC to pass the “hello world” test.

The middle layer routines are used for most applications. RPC calls are made with the system routines registerrpc(), callrpc(), and svc_run().

The first two of these are the most fundamental: registerrpc() obtains a unique system-wide procedure identification number, and callrpc() actually executes a remote procedure call. At the middle layer, a call to rnusers() is implemented by way of these two routines.

The middle layer, however, is rarely used in serious programming due to its simplicity. It does not allow timeout specifications or the choice of transport. It allows no UNIX process control or flexibility in case of errors. It does not support multiple types of call authentication. You rarely need all these types of control, but one or two of them is often necessary.

The Lowest Layer

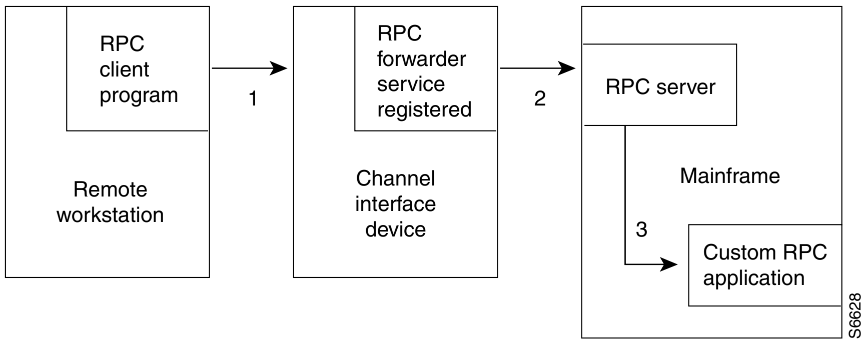
The lowest layer lets you control these details, and for that reason it is often necessary. Programs written at this level are also most efficient, but this is usually not an issue, since RPC clients and servers rarely generate heavy network loads.

Note Although this document discusses only the interface to C, remote procedure calls can be made from any language. Even though this document discusses how RPC is used to communicate between processes on different machines, RPC works just as well for communication between different processes on the same machine.

The RPC Paradigm

Here is a diagram of the RPC paradigm:

Figure 2-1 RPC Paradigm



1 RPC client program issues an RPC call to a remote procedure.

2 RPC forwarder recognizes call to a mainframe service and forwards RPC call over the channel.

3 Mainframe RPC server accepts the call and forwards the request to the RPC application.

Higher Layers of RPC

This section describes the highest and middle layers of RPC.

Highest Layer

Suppose you are writing a program that needs to know how many users are logged into a remote machine. You can do this by calling the RPC library routine rnusers(), as illustrated here:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char **argv;
{
    int num;

    if (argc != 2)
    {
        fprintf(stderr, "usage: rnusers hostname\n");
        exit(1);
    }
    if ((num = rnusers(argv[1])) < 0)
    {
        fprintf(stderr, "error: rnusers\n");
        exit(-1);
    }
    printf("%d users on %s\n", num, argv[1]);
    exit(0);
}
```

RPC Service Library

On a UNIX system, RPC library routines such as rnusers() are in the RPC services library librpcsvc.a. Therefore, the previous program should be compiled on UNIX using this format:

```
example% cc program.c -lrpcsvc
```

Note Cisco IOS for S/390 RPC/XDR does not provide this RPC services library. However, applications may be written on MVS, which accesses these functions on other machines.

Some of the available RPC service library routines are listed below:

Table 2-1 RPC Service Library Routines

Routine	Description
rnusers	Returns number of users on remote machine
rusers	Returns information about users on remote machine
havedisk	Determines if remote machine has disk
rstats	Gets performance data from remote kernel
rwall	Writes to specified remote machines
yppasswd	Updates user password in Yellow Pages

Other RPC services, such as ether(), mount(), and spray(), are not available to the C programmer as library routines. They do, however, have RPC program numbers (which are discussed in the next section), so they can be invoked with callrpc(). Most of the other RPC services also have compilable rpcgen(1) protocol description files.

Note The rpcgen protocol compiler radically simplifies the process of developing network applications. See Chapter 4, Using rpcgen for detailed information about rpcgen and the rpcgen protocol description file. rpcgen is not currently provided with RPC/XDR.

Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions callrpc() and registerrpc(). Using this method, the number of remote users can be found by using this program. It can be used with the RPC/XDR code supplied if the #defines for the ruser program are used.

```
#include <stdio.h>
#include <rpc.h>
/* #include <rusers.h> /* not included with RPC/XDR */

#define RUSERSPROG 10002
#define RUSERSVERS 2
#define RUSERPROC-NUM 1

main(argc, argv)
    int argc; char **argv;
{
    unsigned long nusers;
    int stat;
```

```
        if (argc != 2)
        {
            fprintf(stderr, "usage: (char) nusers hostname\n");
            exit(-1);
        }

        if (stat = callrpc(argv[1], RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
                           xdr_void, 0, xdr_u_long, &nusers) != 0)
           {
               clnt_perrno(stat);
               exit(1);
           }
        printf("%d users on %s\n", nusers, argv[1]);
        exit(0);
    }
```

Unique RPC Procedure Definition

Each RPC procedure is uniquely defined by a program number, version number, and procedure number.

- The program number specifies a group of related remote procedures, each of which has a different procedure number.
- Each program also has a version number, so when a minor change is made to a remote service, a new program number does not have to be assigned.
- Whenever a new procedure is added to a program, it is also given an identifying number.

When you want to call a procedure to find the number of remote users, you look up the appropriate program, version, and procedure numbers in a manual, just as you look up the name of a memory allocator when you want to allocate memory.

The callrpc Library Routine

The simplest way of making remote procedure calls is with the RPC library routine callrpc(). The arguments are:

argv(1)	The name of the remote server machine
RUSERSPROG	The program
RUSERSVERS	The version
RUSERSPROC_NUM	The procedure number. Together with the program and version numbers, this defines the procedure to be called.
xdr_void	An XDR filter
0	An argument to be encoded and passed to the remote procedure
xdr_u_long	A filter for decoding the results returned by the remote procedure
&nusers	A pointer to the place where the procedure's results are to be stored. 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 (cast into an integer) are found in clnt.h.

Since data types may be represented differently on different machines, callrpc() needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result).

For RUSERSPROC_NUM, the return value is an unsigned long so callrpc() has xdr_u_long() as its first return parameter, which says that the result is of type unsigned long and &nusers is its second return parameter, which is a pointer to where the long result is placed. Since RUSERSPROC_NUM takes no argument, the argument parameter of callrpc() is xdr_void().

If callrpc() gets no answer after several tries to deliver a message, it returns an error. The delivery mechanism is User Datagram Protocol (UDP). Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the RPC library, discussed later in this document. The remote server procedure corresponding to the previous example might look like this:

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

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

It takes one argument: a pointer to the input of the remote procedure call (ignored in the example), and it returns a pointer to the result.

Note In the current version of C, character pointers are the generic pointers, so both the input argument and the return value are cast to (char *).

Registering RPC Calls

Normally, a server registers all of the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests. In this example, there is only a single procedure to register, so the main body of the server looks like this:

```
#include <stdio.h>
#include <rpc.h>
#include <rusers.h> /* (not provided with RPC/XDR) */
char *nuser();

main()
{
    registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM, nuser, xdr_void, xdr_u_long);
    svc_run();
    fprintf(stderr, "error: svc_run returned!\n");
    exit(1);
}
```

Note This assumes that a program nuser exists in the RPC library. It is not provided as part of the Cisco IOS for S/390 RPC/XDR.

registerrpc Arguments

The registerrpc() routine registers a C procedure as corresponding to a given RPC procedure number. It has these arguments:

RUSERPROG	The program of the remote procedure to be registered
RUSERSVERS	The version of the remote procedure to be registered
RUSERSPROC_NUM	The procedure number of the remote procedure to be registered
nuser	The name of the local procedure that implements the remote procedure
xdr_void()	XDR filters for the remote procedure's arguments. Multiple arguments are passed as structures.
xdr_u_long()	XDR filters for the remote procedure's results. Multiple results are passed as structures.

Only the UDP transport mechanism can use registerrpc(); thus, it is always safe in conjunction with calls generated by callrpc().

Note The UDP transport mechanism can only deal with arguments and results less than 8K bytes in length.

After registering the local procedure, the server program's main procedure calls svc_run(), the RPC library's remote procedure dispatcher. This function calls the remote procedures in response to RPC call messages. The dispatcher takes care of decoding remote procedure arguments and encoding results, using the XDR filters specified when the remote procedure was registered.

Note Called remote procedures must have the results stored as static variables or external to the called procedure, so that the value is not lost when the procedure is exited.

Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 according to this table:

Table 2-2 Program Numbers

Program Number	Assignment
0x0 - 0x1fffffff	Defined by Sun: Sun Microsystems administers this group of numbers, which should be identical for all Sun customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range.
0x20000000 - 0x3fffffff	Defined by user: This group is reserved for specific customer applications. This range is intended primarily for debugging new programs.
0x40000000 - 0x5fffffff	Transient: This group is reserved for applications that generate program numbers dynamically.
0x60000000 - 0x7fffffff	Reserved for future use; should not be used
0x80000000 - 0x9fffffff	Reserved for future use; should not be used
0xa0000000 - 0xbfffffff	Reserved for future use; should not be used
0xc0000000 - 0xdfffffff	Reserved for future use; should not be used

Table 2-2 Program Numbers (Continued)

Program Number	Assignment
0xe0000000 - 0xffffffff	Reserved for future use; should not be used

To register a protocol specification, or to obtain a complete list of registered programs, send a request by network mail to rpc@sun.com, or write to:

RPC Administrator
Sun Microsystems
2550 Garcia Avenue
Mountain View, CA 94043

When registering a protocol specification, please include a compilable rpcgen ".x" file describing your protocol. You are given a unique program number in return. The RPC program numbers and protocol specifications of standard Sun RPC services can be found in the include files in /usr/include/rpcsvc on most UNIX machines. These services, however, constitute only a small subset of those that have been registered.

Passing Arbitrary Data Types

In the previous example, the RPC call passes a single unsigned long. RPC can handle arbitrary data structures, regardless of byte orders or structure layout conventions used by different machines. RPC does this by always converting the data structures 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 such as xdr_u_long() in the previous example, or a user supplied one.

Built-in Type Routines

XDR has these built-in type routines:

```
xdr_int()
xdr_u_int()
xdr_enum()
xdr_long()
xdr_u_long()
xdr_bool()
xdr_short()
xdr_u_short()
xdr_wrapstring()
xdr_char()
xdr_u_char()
xdr_array()
xdr_bytes()
xdr_double()
xdr_float()
xdr_string()
xdr_union()
xdr_vector()
```

The routine xdr_string() exists, but cannot be used with callrpc() and registerrpc(), which only pass two parameters to their XDR routines. xdr_wrapstring, which has only two parameters, is syntactically correct, and it calls xdr_string(). An example of a user-defined type routine, if you want to send this structure, follows:

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

you would call callrpc() by entering:

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

The xdr_simple() routine is written like this:

```
#include <rpc.h>

xdr_simple(xdrsp, simplep)
    XDR *xdrsp;
    struct simple *simplep;
{
    if (!xdr_int(xdrsp, &simplep->a))
        return (0);
    if (!xdr_short(xdrsp, &simplep->b))
        return (0);
    return (1);
}
```

An XDR routine returns nonzero (true for C) if it completes successfully, and zero otherwise. You will find a complete description of XDR in RFC 1014; only a few implementation examples are given here.

Prefabricated Building Blocks

In addition to the built-in primitives, there are also the prefabricated building blocks:

```
xdr_array()
xdr_bytes()
xdr_reference()
xdr_vector()
xdr_union()
xdr_pointer()
xdr_string()
xdr_opaque()
```

To send a variable array of integers, you might package them as a structure:

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

Then make an RPC call such as:

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

The xdr_varintarr() routine is defined like this:

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

xdr_varintarr Arguments

The xdr_varintarr() routine takes these arguments:

xdr_array	The XDR handle
&arrp->data	A pointer to the array
&arrp->arrlnth	A pointer to the size of the array
MAXLEN	The maximum allowable array size
sizeof(int)	The size of each array element
xdr_int	An XDR routine for handling each array element

If the size of the array is known in advance, you can use xdr_vector(), which serializes fixed-length.

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

XDR always converts quantities to 4-byte multiples when serializing. Thus, if either of the previous examples involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine xdr_bytes, which is like xdr_array except that it packs characters; xdr_bytes has four arguments, similar to the first four arguments of xdr_array. For null-terminated strings, there is also the xdr_string() routine, which is the same as xdr_bytes without the length argument. On select it gets the string length from strlen, and on deserializing it creates a null-terminated string.

A Final Example

This is a final example that calls the previously written xdr_simple() as well as the built-in functions xdr_string() and xdr_reference (which chases 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 You could call xdr_simple() here instead of xdr_reference().

Lowest Layer of RPC

In the examples given in the previous section, RPC takes care of many details automatically. This section shows how to change the defaults by using lower layers of the RPC library. It is assumed that you are familiar with sockets, and with the system/function calls for dealing with them.

There are several occasions when you may need to use lower layers of RPC:

- You may need to use TCP, since the higher layer uses UDP, which restricts RPC calls to 8K bytes of data. Using TCP permits calls to send long streams of data.
- You may want to allocate and free memory while serializing or deserializing with XDR routines. There is no call at the higher level to let you free memory explicitly. (See Memory Allocation with XDR additional information.)
- You may need to perform authentication on either the client or server side, by supplying credentials or verifying them. See Authentication for additional information.

More on the Server Side

This server for the nusers program does the same thing as the one using registerrpc() shown earlier in this chapter, but is written using a lower layer of the RPC package:

```
#include <stdio.h>
#include <rpc.h>
#include <utmp.h>
#include <rusers.h> /* not provided with Cisco IOS for S/390 RPC/XDR */

#define RUSERSPROG 10002
#define RUSERSVERS 2
#define RUSERPROC-NUM 1

main()
{
    SVCXPRT *transp;
    int nuser();
    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL)
    {
        fprintf(stderr, "can't create an RPC server\n");
        exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser, IPPROTO_UDP))
    {
        fprintf(stderr, "can't register RUSER service\n");
        exit(1);
    }
    svc_run(); /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}

nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;

{
    unsigned long nusers;

    switch (rqstp->rq_proc)
    {
        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0))
                fprintf(stderr, "can't reply to RPC call\n");
            return;
        case RUSERSPROC_NUM:
            .
            .  Code here to compute the number of users
            .  and assign it to the variable nusers
            .
            if (!svc_sendreply(transp, xdr_u_long, &nusers))
                fprintf(stderr, "can't reply to RPC call\n");
            return;
        default:
            svcerr_noproc(transp);
            return;
    }
}
```

The Server Gets a Transport Handle

First, the server gets a transport handle, which is used for receiving and replying to RPC messages. registerrpc() uses svcudp_create to get a UDP handle. If you require a more reliable protocol, call svctcp_create instead. If the argument to svcudp_create is RPC_ANYSOCK, the RPC library creates a socket on which to receive and reply to RPC calls. Otherwise, svcudp_create expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of svcudp_create and clnttcp_create (the low-level client routine) must match.

If the user specifies the RPC_ANYSOCK argument, the RPC library routines open a socket. Otherwise they expect the user to do so. The routines svcudp_create and clntudp_create cause the RPC library routines to bind their sockets, if they are not bound already.

A service may choose to register its port number with the local portmapper service. This is done by specifying a non-zero protocol number in svc_register. A client can discover the server's port number by consulting the portmapper on their server's machine. This can be done automatically by specifying a zero port number in clntudp_create or clnttcp_create.

The Server Calls pmap_unset

After creating an SVCXPRT, 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 portmapper's tables.

The Program Number is Associated with the nuser Procedure

Finally, the program number for nusers is associated with the procedure nuser. The final argument to svc_register is normally the protocol in use, which, in this case, is IPPROTO_UDP. Unlike registerrpc(), there are no XDR routines involved in the registration process. Also, registration is done on the program level, rather than the procedure level.

The user routine nuser must call and dispatch the appropriate XDR routines based on the procedure number. These tasks, which registerrpc() handles automatically, are handled by nuser:

- Procedure NULLPROC (currently zero) returns with no results. This can be used as a simple test to detect if a remote program is running.
- There is a check for invalid procedure numbers. If one is detected, svcerr_noproc is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via svc_sendreply. Its first argument is the SVCXPRT handle, the second is the XDR routine, and the third is a pointer to the data to be returned.

Handling an RPC Program that Receives Data

Not illustrated in the previous example is how a server handles an RPC program that receives data. You can add a procedure RUSERSPROC_BOOL, which has an argument nusers, and returns TRUE or FALSE depending on whether there are nusers logged on. This is an example of how it looks:

```
case RUSERSPROC_BOOL:
{
    int bool;
    unsigned nuserquery;

    if (!svc_getargs(transp, xdr_u_int, &nuserquery)
    {
        svcerr_decode(transp);
        return;
    }

.
.  Code to set nusers = number of users
.
if (nuserquery == nusers)
    bool = TRUE;
else
    bool = FALSE;
if (!svc_sendreply(transp, xdr_bool, &bool))
    {
        fprintf(stderr, "can’t reply to RPC call\n");
        return (1);
    }
return;
}
```

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

Memory Allocation with XDR

XDR routines not only process input and output, they also perform memory allocation. This is why the second argument of xdr_array is a pointer to an array, rather than the array itself. If it is NULL, then xdr_array allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider this XDR routine xdr_chararr1, which deals with a fixed array of bytes with length SIZE.

```
xdr_chararr1(xdrsp, chararr)
    XDR *xdrsp;
    char chararr[];
{
    char *p;
    int len;
    p = chararr;
    len = SIZE;
    return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

If space has already been allocated in chararr, it can be called from a server like this:

```
char chararr[SIZE];
svc_getargs(transp, xdr_chararr1, chararr);
```

If you want XDR to do the allocation, you would have to rewrite the routine like this:

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

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

Then 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 being used, the character array can be freed with svc_freeargs. svc_freeargs does not attempt to free any memory if the variable indicating it is NULL.

Note In the routine xdr_finalexample, given earlier, if finalp->string was NULL, then 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. When an XDR routine is called from callrpc(), the serializer part is used. When called from svc_getargs, the deserializer is used. And when called from svc_freeargs, the memory deallocator is used.

When building simple examples like those in this section, a user does not have to worry about the three modes. See RFC 1014 for examples of more sophisticated XDR routines that determine which of the three modes they are in and adjust their behavior accordingly.

The Calling Side

When you use callrpc(), you have no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the layer of RPC that lets you adjust these parameters, consider this code to call the nusers service. This program, as shown, can be run on MVS.

```
#include <stdio.h>
#include <rpc.h>
#include <utmp.h>
#include <netdb.h>
#define RUSERSPROG 10002
#define RUSERSVERS 2
#define RUSERSPROC-NUM 1
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, (char*)&nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS)
    {
        clnt_perror(client, "rpc");
        exit(-1);
    }
    clnt_destroy(client);
    close(sock);
    exit(0);
}
```

The CLIENT Pointer

The low-level version of callrpc() is clnt_call(), which takes a CLIENT pointer rather than a host name. It takes these arguments:

client	A CLIENT pointer
RUSRSPROC_NUM	The procedure number
xdr_void	The XDR routine for serializing the argument
0	A pointer to the argument
xdr_u_long	The XDR routine for deserializing the return value
(char*)&nusers	A pointer to where the return value is placed
total_timeout	The total time, in seconds, to wait for a reply

The CLIENT pointer is encoded with the transport mechanism. The callrpc() routine uses UDP, thus it calls clntudp_create to get a CLIENT pointer. To get Transmission Control Protocol (TCP), you use clnttcp_create.

The clntudp_create() Arguments

The clntudp_create() routine takes five arguments; in this example, they are:

&server_addr	The server address
RUSERSPROG	The program number
RUSERSVERS	The version number
pertry_timeout	A timeout value (between tries)
&sock	A pointer to a socket

Thus, the number of tries is 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, however, only if the RPC library opened it.

If the socket was opened by the user, it stays open. This makes it possible, in cases where multiple client handles are using the same socket, to destroy one handle without closing the socket that other handles are using.

To make a stream connection, the call to clntudp_create is replaced with a call to clnttcp_create.

```
clnttcp_create(&server_addr, prognum, versnum, &sock, inputsize, outputsize);
```

There is no timeout argument; instead, the receive and send buffer sizes must be specified. When the clnttcp_create call is made, a connection is established. All RPC calls using that CLIENT handle would use this connection. The server side of an RPC call using TCP has svcudp_create replaced by svctcp_create.

```
transp = svctcp_create(RPC_ANYSOCK, 0, 0);
```

The last two arguments to svctcp_create are send and receive sizes, respectively. If 0 is specified for either of these, the system chooses a reasonable default.

Useful RPC Features

This section discusses some other aspects of RPC that you may find useful.

Select on the Server Side

If a process is handling RPC requests while performing some other activity, and the other activity involves periodically updating a data structure, the process can set an alarm signal before calling svc_run(). But if the other activity involves waiting on a file descriptor, the svc_run() call will not work. The code for svc_run() should be like this:

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

/* Note: getdtablesize is not provided with RPC/XDR    */

    for (;;)
    {
        readfds = svc_fds;
        switch (select(dtbsz, &readfds, NULL,NULL,NULL))
        {
            case -1:
                if (errno == EINTR)
                    continue;
                perror("select");
                return;
            case 0:
                break;
            default:
                svc_getreqset(&readfds);
        }
    }
}
```

You can bypass svc_run() and call svc_getreqset yourself. All you need to know are the file descriptors of the sockets or sockets associated with the programs you are waiting on. Thus you can have your own select that waits on both the RPC socket and your own descriptors. svc_fds is a bit mask of all the file descriptors that RPC is using for services. It can change every time that any RPC library routine is called, because descriptors are constantly being opened and closed (for example, for connections).

For users who prefer to use generic ECBs for synchronization, mvs_svc_run() may be used. In this case, the RPC server acts the same as if called using svc_run(), but control returns to the caller of mvs_svc_run() when an ECB is posted.

Broadcast RPC

The portmapper is a daemon that converts RPC program numbers into DARPA protocol port numbers. For more information about the portmapper, read RFC 1057.

Note Broadcast RPC is not available in the Cisco IOS for S/390 RPC/XDR.

You cannot broadcast RPC without the portmapper. Here are the main differences between broadcast RPC and normal RPC calls:

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
- Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UDP/IP.
- The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
- All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.
- Broadcast requests are limited in size to the Maximum Transfer Unit (MTU) of the local network. For Ethernet, the MTU is 1500 bytes.

Broadcast RPC Synopsis

```
#include <pmapclnt.h>
    .
    .
    .
enum clnt_stat clnt_stat;
    .
    .
    .
clnt_stat = clnt_broadcast(prognum, versnum, procnum,
    inproc, in, outproc, out, eachresult)
        u_long prognum;                    /* program number */
        u_long versnum;                    /* version number */
        u_long procnum;                    /* procedure number */
        xdrproc_t inproc;            /* xdr routine for args */
        caddr_t in;                        /* pointer to args */
        xdrproc_t outproc;       /* xdr routine for results */
        caddr_t out;             /* pointer to results */
        bool_t (*eachresult)();  /* call with each result */
```

The procedure is called each time a valid result is obtained. It returns a boolean that indicates whether or not the user wants more responses.

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

If done is TRUE, then broadcasting stops and clnt_broadcast returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with RPC_TIMEDOUT.

Batching

In the RPC architecture, clients send a call message and wait for servers to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. This is inefficient if the client does not want or need an acknowledgment for every message sent. It is possible for clients to continue computing while waiting for a response using RPC batch facilities.

Server Batching

RPC messages can be placed in a *pipeline* of calls to a desired server; this is called batching. Batching assumes that:

- Each RPC call in the pipeline requires no response from the server, and the server does not send a response message; and
- The pipeline of calls is transported on a reliable byte stream transport such as TCP/IP.

Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Furthermore, the TCP/IP implementation can buffer up many call messages, and send them to the server in one write system call. This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes, as well as the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually do a nonbatched call to flush the pipeline.

Here is an example of batching. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service (using the TCP/IP transport) may look like this:

```
#include <stdio.h>
#include <rpc.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_))
    {
        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 about error */

                    svcerr_decode(transp);
                    break;
                }
                .
                .  Code here to render the strings
                .
                if (!svc_sendreply(transp, xdr_void, NULL))
                    fprintf(stderr, "can't reply to RPC call\n" );
                break;
            case RENDERSTRING_BATCHED:
                if (!svc_getargs(transp, xdr_wrapstring, &s))
                {
                    fprintf(stderr, "can't decode arguments\n") ;

                    /* We are silent in the face of protocol errors */

                break;
                }
            .
            .  Code here to render strings, but send no reply!
            .
            break;
            default:
                svcerr_noproc(transp);
                return; }

            /* Now free string allocated while decoding arguments */

            svc_freeargs(transp, xdr_wrapstring, &s);
        }
    }
```

The service could have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

Client Batching

For a client to take advantage of batching, the client must perform RPC calls on a TCP/IP-based transport and the actual calls must have these attributes:

- The resulting XDR routine must be zero (NULL).
- The RPC call's timeout must be zero.

Here is an example of a client that uses batching to render multiple strings; the batching is flushed when the client gets a null string (EOF):

```
#include <stdio.h>
#include <rpc.h>
#include <socket.h>
#include <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;
        char buf[1000], *s = buf;
        if ((client = clnttcp_create(&server_addr, WINDOWPROG,
                           WINDOWVERS, &sock, 0, 0)) == NULL)
        {
            error("clnttcp_create");
            exit(-1);
        }
        total_timeout.tv_sec = 0;
        total_timeout.tv_usec = 0;
        while (scanf("%s", s) != EOF)
        {
                clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
                       xdr_wrapstring, &s, NULL, NULL, total_timeout);
                if (clnt_stat != RPC_SUCCESS)
                {
                    clnt_perror(client, "batched rpc");
                    exit(-1);
                }
        }

        /* Now flush the pipeline */

        total_timeout.tv_sec = 20;
        clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
            xdr_void, NULL, total_timeout);
        if (clnt_stat != RPC_SUCCESS)
        {
            clnt_perror(client, "rpc");
            exit(-1);
        }
        clnt_destroy(client);
        exit(0);
    }
```

Since the server sends no message, the clients cannot be notified of any of the failures that may occur. Therefore, clients are on their own when it comes to handling errors.

The previous example was completed to render all of the (2000) lines in the UNIX file /etc/termcap. The rendering service did nothing but throw the lines away. The example was run in these configurations, with the indicated results:

Table 2-3 Client Batching Configuration Results

Configuration	Result
Machine to itself, regular RPC	50 seconds
Machine to itself, batched RPC	16 seconds
Machine to machine, regular RPC	52 seconds
Machine to machine, batched RPC	10 seconds

Running fscanf on the UNIX file /etc/termcap only requires six seconds. These timings show the advantage of protocols that allow for overlapped execution, though these protocols are often hard to design.

Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network file system, require stronger security than what has been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type is none.

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support.

UNIX Authentication

The Cisco IOS for S/390 RPC/XDR supports UNIX authentication.

The Client Side

When a caller creates a new RPC client handle by using:

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

the appropriate transport instance defaults the associate authentication handle to:

```
clnt->serverl_auth = authnone_create()
```

The RPC client can choose to use UNIX style authentication by setting the field clnt->serverl_auth after creating the RPC client handle using:

```
clnt->serverl_auth = authunix_create_default()
```

This causes each RPC call associated with clnt to carry an authentication credentials structure:

```
.
. UNIX style credentials
.
struct authunix_parms
{
    u_long   aup_time;          /* credentials creation time */
    char    *aup_machname;      /* host name where client is */
    int      aup_uid;           /* client's 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 */
};
```

These fields are set by authunix_create_default by using the appropriate system calls. Since the RPC user created this new style of authentication, the user is responsible for destroying it with

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

This should be done in all cases to conserve memory.

The Server Side

Service implementors have a harder time dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine using this code:

```
.
.  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 of interest: the style or flavor of authentication credentials, as illustrated here:

```
.
.  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 makes these guarantees to the service dispatch routine:

- The request's rq_cred is well formed. Thus the service implementor may inspect the request's rq_cred.oa_flavor to determine which style of authentication the caller used. The service implementor may also wish to inspect the other fields of rq_cred if the style is not one of the styles supported by the RPC package.
- The request's rq_clntcred field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. Remember that only UNIX style is currently supported, so (currently) rq_clntcred could be cast to a pointer to an authunix_parms structure. If rq_clntcred is NULL, the service implementor may wish to inspect the other (opaque) fields of rq_cred, in case the service knows about a new type of authentication that the RPC package does not know about.

This remote users service example can be extended so that it computes results for all users except UID 16.

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authunix_parms *unix_cred;
    int uid;
    unsigned long nusers;   /* we don't care about authentication
                              for null proc */
    if (rqstp->rq_proc == NULLPROC)
    {
        if (!svc_sendreply(transp, xdr_void, 0))
        {
            fprintf(stderr, "can't reply to RPC call\n" );
            return (1);
        }
        return;
    }
/* now get the uid */

    switch (rqstp->rq_cred.oa_flavor)
    {
        case AUTH_UNIX:
          unix_cred = (struct authunix_parms *)rqstp->rq_clntcred;
          uid = unix_cred->aup_uid;
          break;
        case AUTH_NULL:
        default:
            svcerr_weakauth(transp);
            return;
    }
    switch (rqstp->rq_proc)
    {
        case RUSERSPROC_NUM:   /* make sure caller is allowed
                                 to call this proc */
            if (uid == 16)
            {
                svcerr_systemerr(transp);
                return;
            }
            .
            .  Code here to compute the number of users
            .  and assign it to the variable nusers
            .
            if (!svc_sendreply(transp, xdr_u_long, &nusers))
            {
                fprintf(stderr, "can't reply to RPC call\n" );
                return (1);
            }
            return;
        default:
            svcerr_noproc(transp);
            return;
    }
}
```

A few things should be noted:

- It is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero).
- If the authentication parameter's type is not suitable for your service, you should call svcerr_weakauth.
- The service protocol itself should return status for access denied; in the case of the example, the protocol does not have such a status, so the service primitive svcerr_systemerr is called instead.
- The last point underscores the relation between the RPC authentication package and the services; RPC deals only with authentication and not with individual services' access control. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

DES Authentication

UNIX authentication is quite easy to defeat. Instead of using authunix_create_default, you can call authunix_create and then modify the RPC authentication handle it returns by filling in whatever user ID and host name you want the server to think it has. DES authentication is thus recommended if you want more security than UNIX authentication offers.

Note The Cisco IOS for S/390 RPC/XDR does not currently support DES authentication. Users may write their own DES authorization handler. The details of the DES authentication protocol are complicated and are not explained here. See RFC 1057 for the details.

For DES authentication to work, the keyserv(8c) daemon must be running on both the server and client machines. The users on these machines need public keys assigned by the network administrator in the publickey(5) database. They also need to have decrypted their secret keys using their login password. This happens automatically when you log in using login(1), or you can do it manually using keylogin(1).

Client Side

If a client wishes to use DES authentication, it must set its authentication handle appropriately. Here is an example:

```
cl->cl-auth=
    authdes_create(servername, 60, &server_addr, NULL);
```

The first argument is the network name or "netname" of the owner of the server process. Typically, server processes are root processes and their netname can be derived using this call:

```
char servername[MAXNETNAMELEN];
host2netname(servername, rhostname, NULL);
```

Here, rhostname is the host name of the machine where the server process is running. host2netname fills in servername to contain this root process's netname. If the server process was run by a regular user, you could use the call user2netname instead. Here is an example for a server process with the same user ID as the client:

```
char servername[MAXNETNAMELEN];
user2netname(servername, getuid(), NULL);
```

The last argument to both user2netname and host2netname is the name of the naming domain where the server is located. The NULL used here means "use the local domain name".

authdes_create Arguments

The authdes_create routine takes these arguments:

servername	The name of the server
60	The lifetime of the credential Here it is set to sixty seconds. This means that the credential expires 60 seconds from now. If some mischievous user tries to reuse the credential, the server RPC subsystem recognizes that it has expired and will not grant any requests. If the same user tries to reuse the credential within the sixty second lifetime, the user is still rejected because the server RPC subsystem remembers which credentials it has already seen in the near past, and does not grant requests to duplicates.
&server_addr	The address of the host to which it synchronizes / For DES authentication to work, the server and client must agree on the time. Here the address of the server itself is passed, so the client and server are both using the same time: the server's time. The argument can be NULL, which means "don't bother synchronizing". You should only do this if you are sure the client and server are already synchronized.

NULL	The address of a DES encryption key to use for encrypting time stamps and data / If this argument is NULL, as it is in this example, a random key is chosen. The client may find out the encryption key being used by consulting the ah_key field of the authentication handle.

Server Side

The server side is a lot simpler than the client side. Here is the previous example rewritten to use AUTH_DES instead of AUTH_UNIX:

```
#include <time.h>
#include <authdes.h>
.
.
.
nuser(rqstp, transp)
    struct svc_req *rqstp;

    SVCXPRT *transp;
{
    struct authdes_cred *des_cred;
    int uid;
    int gid;
    int gidlen;
    int gidlist[10];[
/* we don't care about authentication for null proc */
    if (rqstp->rq_proc == NULLPROC)
    {
        .
        .  same as before
        .
    }
/* now get the uid     */
    switch (rqstp->rq_cred.oa_flavor)
    {
        case AUTH_DES:
        des_cred =
          (struct authdes_cred *) rqstp->rq_clntcred;
        if (! netname2user(des_cred->adc_fullname.name, &uid,              &gid,
&gidlen, gidlist))
        {
            fprintf(stderr, "unknown user: %s", des_cred->adc_fullname.name);
            svcerr_systemerr(transp);
            return;
        }
        break;
        case AUTH_NULL:
        default:
            svcerr_weakauth(transp);
            return;
    }
        .
        .  The rest is the same as before
        .
}
```

Notice the use of the routine netname2user, the inverse of user2netname: it takes a network ID and converts to a UNIX ID. netname2user also supplies the group IDs that are not used in this example, but which may be useful to other UNIX programs.

Using Inetd

Note inetd is not available in the Cisco IOS for S/390 RPC/XDR.

An RPC server can be started from inetd.

When starting an RPC server from inetd, the only difference from the usual code is that the service creation routine should be called in this form, since inet passes a socket as file descriptor 0.

```
transp = svcudp_create(0);       /* For UDP */
transp = svctcp_create(0,0,0);   /* For listener tcp sockets */
transp = svcfd_create(0,0,0);    /* For connected tcp sockets */
```

Also, svc_register should be called with the final flag as 0, since the program would already be registered by inetd.

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

Remember, if you want to exit from the server process and return control to inet, you need to explicitly exit, since svc_run() never returns.

The format of entries in /etc/inetd.conf for RPC services can be either one of these:

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

p_name	The symbolic name of the program as it appears in rpc(5)
server	The program implementing the server
version	The version number of the service

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
```

Programming Examples

This section presents more examples of remote procedure calls.

Versions

By convention, the first version number of program PROG is PROGVERS_ORIG; the most recent version is PROGVERS. For a new version of the user program named RUSERSVERS_SHORT that returns an unsigned short rather than a long, a server that wants to support both versions would do a double register.

```
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, user,
                  IPPROTO_tcp))
{
    fprintf(stderr, "can't register RUSER service\n");
    exit(1);
}
```

Both versions can be handled by the same C procedure, as this example illustrates:

```
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2;

    switch (rqstp->rq_proc)
    {
        case NULLPROC:
            if (!svc_sendreply(transp, xdr_void, 0))
            {
                fprintf(stderr, "can't reply to RPC call\n");
                return (1);
            }
            return;
        case RUSERSPROC_NUM:
        .
        .   Code here to compute the number of users
        .   and assign it to the variable nusers
        .
            nusers2 = nusers;
            switch (rqstp->rq_vers)
            {
                case RUSERSVERS_ORIG:
                    if (!svc_sendreply(transp, xdr_u_long,
                        &nusers))
                    {
                      fprintf(stderr,"can't reply to RPC call\n");
                    }
                    break;
                case RUSERSVERS_SHORT:
                    if (!svc_sendreply(transp, xdr_u_short,
                         &nusers2))
                    {
                      fprintf(stderr,"can't reply to RPC call\n");
                    }
                    break;
            }
            default:
                svcerr_noproc(transp);
                return;
        }
}
```

TCP

Here is an example that is essentially rcp, a UNIX remote file copy program that copies files between machines. The initiator of the RPC snd call takes its standard input and sends it to the server rcv, which prints it on standard output. The RPC call uses TCP. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

```
/* The xdr routine: on decode, read from wire, write onto fp
 * on encode, read from fp, write onto wire */

#include <stdio.h>
```

```
#include <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)
        {
            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.h>
#include <socket.h>
#include <time.h>
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 sockets, = RPC_ANYSOCK;
    enum clnt_stat clnt_stat;
    struct hostent *hp;
    register CLIENT *client;
    struct timeval total_timeout;

    if ((hp = gethostbyname(host)) == NULL)
    {
        fprintf(stderr, "can't get addr for '%s'\n", host);
        return (-1);
    }
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,              hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = 0;
    if ((client = clnttcp_create(&server_addr, prognum,
         versnum, &sockets,, BUFSIZ, BUFSIZ)) == NULL)
    {
        perror("rpctcp_create");
        return (-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum, inproc, in, outproc,                 out,
total_timeout);
    clnt_destroy(client);
    return (int)clnt_stat;
}

/* The receiving routines     */

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

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 (1);
                }
                return;
            case RCPPROC_FP:
                if (!svc_getargs(transp, xdr_rcp, stdout))
                {
                    svcerr_decode(transp);
                    return;
                }
                if (!svc_sendreply(transp, xdr_void, 0))
                {
                    fprintf(stderr, "can't reply\n");
                    return;
                }
                return (0);
            default:
                svcerr_noproc(transp);
                return;
        }
    }
```

Callback Procedures

Occasionally, it is useful to have a server become a client and make an RPC call back to the process that is its client. An example is remote debugging, where the client is a window system program and the server is a debugger running on the remote machine. Most of the time the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a breakpoint, the roles are reversed, and the debugger wants to make an RPC call to the window program, so that it can inform the user that a breakpoint has been reached.

To do an RPC callback, you need a program number on which to make the RPC call. Since this is a dynamically generated program number, it should be in the transient range, 0x40000000 - 0x5fffffff. The routine gettransient returns a valid program number in the transient range, and registers it with the portmapper. It only talks to the portmapper running on the same machine as the gettransient routine itself. The call to pmap_set is a test and set operation, because it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it. On return, the sockp argument points to a socket, that can be used as the argument to an svcudp_create() or svctcp_create() call.

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

gettransient(proto, vers, sockp)
    int proto, vers, *sockp;
{
    static int prognum = 0x40000000;
    int s, len, socktype;
    struct sockaddr_in addr;

    switch(proto)
```

```
        {
            case IPPROTO_UDP:
                socktype = SOCK_DGRAM;
                break;
            case IPPROTO_TCP:
                socktype = SOCK_STREAM;
                break;
            default:
                fprintf(stderr, "unknown protocol type\n");
                return 0;
        }
    if (*sockp == RPC_ANYSOCK)
        {
            if ((s = sockets,(AF_INET, socktype, 0)) < 0)
            {
                perror("sockets,");
                return (0);
            }
            *sockp = s;
        }
    else
        s = *sockp;
        addr.sin_addr.s_addr = 0;
        addr.sin_family = AF_INET;
        addr.sin_port = 0;
        len = sizeof(addr);

        /* may be already bound, so don't check for error */

        bind(s, &addr, len);
        if (getsockname(s, &addr, &len)< 0)
        {
            perror("getsockname");
            return (0);
        }
        while (!pmap_set(prognum++, vers, proto, ntohs(addr.sin_port))) continue;
        return (prognum-1);
    }
```

Note The call to ntohs is necessary to ensure that the port number in addr.sin_port, which is in network byte order, is passed in host byte order (as pmap_set expects).

The following client and server programs illustrate how to use the gettransient routine. The client makes an RPC call to the server, passing it a transient program number. Then the client waits around 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. Then 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.

Client

The client program:

```
/* client    */

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

int callback,();
char hostname[256];
```

```
main()
{
    int x, ans, s;
    SVCXPRT *xprt;

    gethostname(hostname, sizeof(hostname));
    s = RPC_ANYSOCK;
    x = gettransient(IPPROTO_UDP, 1, &s);
    fprintf(stderr, "client gets prognum %d\n", x);
    if ((xprt = svcudp_create(s)) == NULL)
    {
        fprintf(stderr, "rpc_server: svcudp_create\n");
        exit(1);
    }

    /* protocol is 0 - gettransient does registering */

    (void)svc_register(xprt, x, 1, callback,, 0);
    ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
                  EXAMPLEPROC_callback, xdr_int, &x, xdr_void, 0);
    if ((enum clnt_stat) ans != RPC_SUCCESS)
    {
        fprintf(stderr, "call: ");
        clnt_perrno(ans);
        fprintf(stderr, "\n");
    }
    svc_run();
    fprintf(stderr, "error: svc_run shouldn't return\n");
}
callback,(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc)
    {
        case 0:
            if (!svc_sendreply(transp, xdr_void, 0))
            {
                fprintf(stderr, "err: exampleprog\n ");
                return (1);
            }
            return (0);
        case 1:
            if (!svc_getargs(transp, xdr_void, 0))
            {
                svcerr_decode(transp);
                return (1);
            }
            fprintf(stderr, "client got callback,\n");
            if (!svc_sendreply(transp, xdr_void, 0))
            {
                fprintf(stderr, "err: exampleprog") ;
                return (1);
            }
    }
}
```

Server

The server program:

```
/* server     */
#include <stdio.h>
#include <rpc.h>
#include <signal.h>

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

main()
{
    gethostname(hostname, sizeof(hostname));
    registerrpc(EXAMPLEPROG, EXAMPLEVERS,
                EXAMPLEPROC_callback,, getnewprog, xdr_int, xdr_void);
    fprintf(stderr, "server going into svc_run\n");
    signal(SIGALRM, docallback,);
    alarm(10);
    svc_run();
    fprintf(stderr, "error: svc_run shouldn't return\n");
}

char *
getnewprog(pnump)
    char *pnump;
{
    pnum = *(int *)pnump;
    return NULL;

}
docallback,()
{
    int ans;

    ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0, xdr_void, 0);
    if (ans != 0)
    {
         fprintf(stderr, "server: ");
         clnt_perrno(ans);
         fprintf(stderr, "\n");
    }
}
```