Remote Procedure Call (RPC) Programming [2]

This section describes various aspects of remote procedure call (RPC) programming and provides examples of its use.

Although the examples illustrate the interface to the C programming language only, RPCs can be made from any language. Examples show RPC programming as it is used to communicate between processes on various machines, but the procedure is the same for communication between different processes on the same machine.

Typically, using RPC consists of registering the routine that will be accessed, making the request for the registered routine to perform its function, and passing values between the registered routine and the calling routine. The examples in this section show how you can accomplish this. Following is an example of a typical RPC procedure:

Example 1:

A server registers a program that will calculate the factorial of an integer and will return the square root of the factorial. A client program accepts as input an integer value and then makes an RPC to the server, passing it the integer value. The server performs the calculations and returns the answer. The return type is `double`.

For more information on RPC programming, see appendix B, page 97, appendix C, page 121, and appendix D, page 141.

Subsections 2.1, page 12, and 2.2, page 14, provide code for and explanations of these processes.

Registering the routine on the server

2.1

The server registers the routine that will be used to do the computation and then exits into a service loop to wait for requests. The server does not use any CPU resources while waiting for requests.

Example 1A contains all of the code needed to perform the server function. This code is entirely portable in the sense that it can run on a Cray Research system or another system anywhere on the network. In fact, any machine on the network that supports RPCs (as well as sockets, UDP, TCP, and a C compiler) can run this server.

Example 1A:

```
/*
 *     This is the server routine for example 1
 */

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

#define PROGRAM 0x20000100
#define VERSION 1
#define ROUTINE 1

extern  double  sqrt();
double  *compute_result();

main()
{
       if(registerrpc(PROGRAM,VERSION,ROUTINE,compute_result,
                        xdr_int, xdr_double) == –1)
        {
                perror(“registerrpc”);
                exit(1);
        }
        svc_run();
        fprintf(stderr,“svc_run() call failed\n”);
        exit(1);
}

double  *
compute_result(input)
int     *input;
{
int     count;
static  double  output;

        output=1.0;
        for(count= *input; count>1; count--)
                output *= count;

        output = sqrt(output);
        return(&output);
}
```

The following text explains the RPC portions of the server source code in example 1A.

Line 6: If XDR routines are being used, the `<rpc/rpc.h>` include file is always necessary. Two XDR routines are used in line 18. (See a discussion of XDR routines in subsection 1.1, page 2.)

Lines 8 through 10: Constants `PROGRAM`, `VERSION`, and `ROUTINE` uniquely define the RPC being registered. (See a discussion of these constants in subsection 1.2.2, page 6, and subsection 1.2.3, page 8.) All three of the constants are parameters in the `registerrpc` call made in line 17.

Line 17: This is the call that registers the RPC with the portmapper process so that other programs on the network can find it. The parameters are as follows: program number (`PROGRAM`), version number (`VERSION`), routine number (`ROUTINE`), name of routine associated with routine number (`compute_result`), data translation routine for incoming value (`xdr_int`), and data translation routine for return value (`xdr_double`).

Line 23: This is the exit into the service loop. The server can, of course, call other routines or do any required setup before calling the `svc_run` routine. However, client requests cannot be processed until `svc_run` is called.

Line 33: It is critical that the variable containing the returned value be static; otherwise, it might disappear by the time RPC/XDR sends it out in the response packet.

Client call and server reply process
2.2

In example 1B, the client receives an input value and passes it to the server by using an RPC. The server computes a result and returns it to the client, where it is then printed out.

Example 1B:

```
/*
 *      This is the client routine for example 1
 */

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

#define PROGRAM 0x20000100
#define VERSION 1
#define ROUTINE 1

main(argc, argv)
int     argc;
char    **argv;
{
int     input,
        ret_val;
double  result;
char    input_buf[25];

        printf(“Enter an Integer=>”);
        fflush(stdout);
        fgets(input_buf,25,stdin);
        input = atoi(input_buf);
       if((ret_val=callrpc(argv[1],PROGRAM,VERSION,ROUTINE,
                        xdr_int, &input, xdr_double, &result))
                != 0)
        {
                clnt_perrno(ret_val);
                exit(1);
        }
        printf(“Result = %E\n”,result);
}
```

The following text explains the RPC portions of the client source code in example 1B.

Line 25: This is the actual call to the server. The client routine is given the host name on the command line. The parameters to the `callrpc` routine are as follows:

- Network name of the host on which the server is running
- Program number (`PROGRAM`)
- Version number (`VERSION`)
- Routine number (`ROUTINE`)
- XDR translation routine for the variable being passed to the server (`xdr_int`)
- Source address of the variable being passed to the server (`input`)
- XDR translation routine for the variable being returned from the server (`xdr_double`)
- Destination address of the result being returned from the server (`result`)

Lines 29, 30: This is the RPC client error routine. You can diagnose failure of certain RPC routines through the return value of the failing routine. For example, if the client were executed and the specified server host were not running, the following error message would be returned:

```
RPC: Program not registered
```

RPC layers
2.3

The RPC interface is divided into three layers. The highest layer is totally transparent to programmers. To illustrate, at this level, a program can contain a call to routine `rnusers`(3), which returns the number of users on a remote machine. You do not have to be aware that an RPC interface is being used, because you simply make the call in a program, just as you would call `malloc`(3).

At the intermediate layer, routines `registerrpc` and `callrpc` are used to make RPCs; `registerrpc` obtains a number that is unique across the system, while `callrpc` executes an RPC. The `rnusers`(3) call is implemented by the use of these two routines. The intermediate-layer routines are designed for most common applications.

The lowest layer is for more sophisticated applications, such as altering the defaults of the routines. At this layer, you can explicitly manipulate the sockets that transmit RPC messages.

Highest RPC layer
2.3.1

Imagine you are writing a program to determine how many users are logged on to a remote machine. You can do this by calling routine `rnusers`(3), as shown in example 2.

Example 2:

```
#include <stdio.h>
main(argc, argv)
    int argc;
    char **argv;
{
    unsigned num;
    if (argc <2) {
        fprintf(stderr, “usage: rnusers hostname\n”);
        exit(1);
    }
    if ((num = rnusers(argv[1])) <0) {
        fprintf(stderr, “error: rnusers\n”);
        exit(–1);
    {
    printf(”%d users on %s\n”, num, argv[1]);
    exit(0);
}
```

RPC library routines such as `rnusers`(3) are in the RPC services library, `librpcsvc.a`. Thus, you should use the following command to compile the program in example 3 on Cray Research systems:

```
% cc program.c –lrpcsvc
```

The `rnusers` routine and other RPC library routines are documented in appendix F, page 177. Table 3 lists RPC service library routines available to C programmers. These routines are supported only on the client side. You can invoke the other RPC services (`ether`, `mount`, `rquota`, and `spray`), which are not available to C programmers as library routines, by using the `callrpc` routine, as described in subsection 2.3.2.

Table 3. RPC service library routines

Routines	Description
`getpublickey`	Gets public key
`getrpcport`	Gets RPC port number
`getsecretkey`	Gets secret key
`havedisk`	Determines whether remote machine has a disk
`rnusers`	Returns number of users on remote machine
`rstat`	Gets performance data from remote kernel
`rusers`	Returns information about users on remote machine
`rwall`	Writes to specified remote machines
`yppasswd`	Updates user password in the NIS database

Intermediate RPC layer
2.3.2

Instead of calling routine `rnusers` as shown in example 3, you can use functions `registerrpc` and `callrpc` to make the `rnusers` call, as illustrated in examples 3 and 4. These functions use the UDP transport mechanism, whose arguments and results are constrained by the maximum length of UDP packets. Consult the vendor documentation for exact length restrictions.

2.3.2.1 *Registering in the intermediate layer*

Usually, a server registers all RPCs it plans to handle and then goes into an infinite loop, waiting to service requests. In the main body of the server routine, you can register only one procedure, as shown in example 3.

Example 3:

```
1      #include <stdio.h>
2      #include <rpcsvc/rusers.h>
3      char *nuser();
4      main()
5      {
6          registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
7              nuser, xdr_void, xdr_u_long);
8          svc_run();      /* never returns */
9          fprintf(stderr, "Error: svc_run returned!\n");
10         exit(1);
11     }
12
13     char *
14     nuser(indata)
15         char *indata;
16     {
17         static int nusers;
18         /*
19          * code here to compute the number of users
20          * and place result in variable nusers
21          */
22         return((char *)&nusers);
23     }
```

The following text explains the RPC portion of the server source code in example 3.

Lines 6 and 7: The `registerrpc` routine matches each RPC procedure number with a C procedure. The first three parameters, `RUSERSPROG`, `RUSERSVERS`, and `RUSERSPROC_NUM`, are the program, version, and procedure numbers of the remote procedure to be registered; `nuser()` is the name of the C procedure implementing it; and `xdr_void` and `xdr_u_long` are, respectively, the types of the input to and output from the procedure.

Calling and replying in the intermediate layer 2.3.2.2

Example 4 shows the client source code used in the intermediate layer.

Example 4:

```
 1      #include <stdio.h>
 2      #include <rpcsvc/rusers.h>
 3      main(argc, argv)
 4          int argc;
 5          char **argv;
 6      {
 7          unsigned long nusers;
 8          if (argc < 2) {
 9              fprintf(stderr, “usage: nusers hostname\n”);
10             exit(–1);
11         }
12         if (callrpc(argv[1],
13          RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
14          xdr_void, NULL, xdr_u_long, &nusers) != NULL) {
15             fprintf(stderr, “error: callrpc\n”);
16             exit(1);
17         }
18         printf(”%d users on %s\n”, nusers, argv[1]);
19         exit(0);
20     }
```

The following text explains the RPC portion of the client source code in example 4.

Lines 12 through 16: The `callrpc` RPC library routine has eight parameters. The first is the name of the remote machine (`argv[1]`). The next three parameters are the program (RUSERSPROG), version (RUSERSVERS), and procedure numbers (RUSERSPROC_NUM).

Because you can represent data types differently on various machines, `callrpc` requires both the type of the RPC argument and a pointer to the argument itself (and, similarly, a type and pointer for the result). Because the remote procedure requires no argument, the input data type parameter of `callrpc` is `xdr_void`. The first return parameter is `xdr_u_long`, which indicates that the result is of type `unsigned long`. The second return parameter is `&nusers`, which is a pointer to the destination of the type `long` result.

Lines 10, 16, and 19: If it completes successfully, `callrpc` returns a 0; otherwise, it returns a nonzero value. The exact meaning of the return codes is found in file `<rpc/clnt.h>`, and is in fact an enumeration cast into an integer (type defined as `clnt_stat`).

If `callrpc` gets no answer after trying several times to deliver a message, it returns with an error code. The delivery mechanism is UDP. Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the RPC library, discussed in subsection 2.3.3, page 26.

Using XDR routines
2.3.2.3

In example 3, the RPC passes one value of type `unsigned long`. RPC handles arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by converting them to a network standard called External Data Representation (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called *serializing*; the reverse process is called *deserializing*. The type field parameters of `callrpc` and `registerrpc` can specify a built-in procedure (such as `xdr_u_long` in example 3) or a user-supplied one. XDR has the following built-in type routines:

```
xdr_bool()        xdr_u_char()
xdr_char()        xdr_u_int()
xdr_enum()        xdr_u_long()
xdr_int()         xdr_u_short()
xdr_long()        xdr_void()
xdr_short()       xdr_wrapstring()
```

An XDR routine returns a nonzero value (`TRUE` in the context of C) if it completes successfully; otherwise, it returns a 0.

In addition to the built-in type routines, the following prefabricated building blocks also exist:

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

Several of these routines are described in the following paragraphs. All of them are described in appendix A, page 67.

To send a variable-length array of integers, you could package them as a structure, as follows:

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

You could then make the following RPC:

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

The `xdr_varintarr()` routine is defined, as follows:

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

The `xdr_array` routine takes as parameters the XDR handle (`xdrsp`), a pointer to the array (`&arrp->data`), a pointer to the size of the array (`&arrp->arrlnth`), the maximum allowable array size (`MAXLEN`), the size of each array element (`sizeof(int)`), and an XDR routine for handling each array element (`xdr_int`).

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

To send out an array of `SIZE` integers, you could use the following routine:

```
int int_array[SIZE];
xdr_intarr(xdrsp, intarr)
    XDR *xdrsp;
    int intarr[];
{
    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. The `xdr_bytes` routine has four parameters, which are similar to the first four parameters of `xdr_array`. For null-terminated strings, there is also the `xdr_string` routine, which is the same as `xdr_bytes` without the length parameter. On serializing, `xdr_string()` gets the string length from `strlen()`; on deserializing, it creates a null-terminated string.

The following code shows a user-defined type routine in which you send the structure

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

and call `callrpc`, as follows:

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

Write `xdr_simple()`, as follows:

```
#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);
}
```

Example 5 calls the previously written `xdr_simple()`, as well as the built-in functions `xdr_string` and `xdr_reference`, to dereference pointers.

Example 5:

```
typedef 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);
}
```

By using `xdr_reference` instead of merely calling `xdr_simple()`, you yield the burden of allocating and freeing storage for the referenced structure to the RPC library. If `xdr_simple()` were used, you would be forced to provide code for these memory management functions.

XDR memory allocation 2.3.2.4

Besides performing input and output operations, XDR routines also perform memory allocation. This is why the second parameter of `xdr_array` is a pointer to an array, rather than the array itself. If the second parameter is `NULL`, `xdr_array` allocates space for the array and returns a pointer to it, putting the size of the array in the third parameter. As an example, consider the following XDR routine, `xdr_chararr1()`, which deals with a fixed array of bytes with length `SIZE`.

```
xdr_chararr1(xdrsp, chararr)
    XDR *xdrsp;
    char chararr[];
{
    char *p;
    int len;
```

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

It might be called from a server, as follows:

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

In this case, `chararr` has already allocated space. If you want XDR to do the allocation, you must rewrite this routine in the following way:

```
xdr_chararr2(xdrsp, chararrp)
    XDR *xdrsp;
    char **chararrp;
{
    int len;
    len = SIZE;
    return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

Then the RPC 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 the character array has been used, you can free it by using `svc_freeargs`. In the `xdr_finalexample()` routine shown in example 5, imagine that `finalp->string` was `NULL` in the following call:

```
svc_getargs(transp, xdr_finalexample, &finalp);
```

The `svc_getargs` call is described in the following subsection. To free the array allocated to hold `finalp->string`, you could issue the following call:

```
svc_freeargs(xdrsp, xdr_finalexample, &finalp);
```

If `finalp->string` is `NULL`, this call frees nothing. The same is true for `finalp->simplep`.

To summarize, each XDR routine is responsible for serializing, deserializing, and allocating memory. When an XDR routine is called from `callrpc`, the serializer is used; when the routine is called from `svc_getargs`, the deserializer is used; when it is called from `svc_freeargs`, the memory deallocator is used.

Lowest RPC layer 2.3.3

In the high and intermediate layers, RPC handles many details automatically for you. This subsection explains how you can change the defaults of routines by using the lowest layer of the RPC library. It is assumed that you are familiar with sockets and the system calls for dealing with them. If you are not, see `socket`(2).

You can use the lowest layer of RPC under various conditions. First, you might need to use TCP. The higher and intermediate layers use UDP, which might restrict RPCs to 8 Kbytes of data. Using TCP permits calls to send long streams of data (for an example, see subsection 2.3.3.4, page 34). Second, you might want to allocate and free memory while serializing or deserializing with XDR routines. No call at the higher or intermediate level exists to let you free memory explicitly (for more explanation, see subsection 2.3.2.4, page 24). Third, you might need to perform authentication on either the client or server side by supplying credentials or verifying them (see the explanation in section 3, page 53).

Registering in the lowest layer 2.3.3.1

The server for the `nusers` program shown in example 6 uses a lower layer of the RPC package but performs the same function as the server in example 3, which uses `registerrpc`.

Example 6:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
main()
{
    SVCXPRT *transp;
    int nuser();
    transp=svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){
        fprintf(stderr, “can’t create an RPC server\n”);
        exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS,
            nuser, IPPROTO_UDP)) {
        fprintf(stderr, “can’t register RUSER service\n”);
        exit(1);
    }
    svc_run(); /* never returns */
    fprintf(stderr, “should never reach this point\n”);
}
nuser(rqstp, tranp)
    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;
        }
        return;
    case RUSERSPROC_NUM:
        /*
         * code here to compute the number of users
         * and put in variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers) {
            fprintf(stderr, “can’t reply to RPC call\n”);
            return;
        }
        return;
    default:
        svcerr_noproc(transp);
        return;
    } }
```

The following text explains the RPC portions of the server source code in example 6.

Lines 6 through 11: First, the server gets a transport handle, which is used for sending out and replying to RPC messages. This example uses `svcudp_create` to get a UDP handle. If you require a reliable protocol, call `svctcp_create` instead. If the argument to `svcudp_create` is `RPC_ANYSOCK` (as in the example), the RPC library creates a socket on which to send out RPCs; 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, the port numbers of `svcudp_create` and `clntudp_create` (the low-level client routines) must match.

When you specify `RPC_ANYSOCK` for a socket or give an unbound socket, the system determines port numbers in the following way:

1. When a server starts up, it advertises to a portmapper daemon on its local machine.
2. The server-side portmap daemon picks a port number for the RPC procedure if the socket specified as a parameter to `svcudp_create` is not already bound.
3. On the client side, when the `clntudp_create` call is made with an unbound socket, the system queries the portmapper on the machine to which the call is being made, and it gets the appropriate port number.
4. If the portmapper is not running on the server side, or has no port that corresponds to the RPC, the RPC fails.

You can make RPCs to the portmapper yourself. The appropriate procedure numbers are in include file `<rpc/pmap_prot.h>`.

Lines 13 through 17: After creating a service transport handle, `(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.

Finally, the program number for `nusers` is associated with the `nuser` routine. The final argument to `svc_register` is usually the protocol being used, which, in this case, is `IPPROTO_UDP`. Notice that, unlike `registerrpc`, no XDR routines are involved in this registration process. Also, registration is done on the program, rather than procedure, level.

Lines 28 through 46: The `nuser` routine must call and dispatch the appropriate XDR routines, based on the procedure number.

The `nuser` routine handles three conditions. First, procedure `NULLPROC` (currently 0) returns without arguments. You can use this as a simple test for detecting whether a remote program is running. Second, `nuser` checks for valid procedure numbers. Third, `svcerr_noproc`, which is the default, is called to handle the error.

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

Not illustrated in example 6 is how a server handles an RPC program that passes data. In example 7, a procedure, `RUSERSPROC_BOOL`, is added. This procedure has an argument, `nusers`, and returns `TRUE` or `FALSE` if the number of users logged on equals the number specified by `nusers`. The relevant routine is `svc_getargs`, which takes an `SVCXPRT` handle, the XDR routine, and a pointer to the destination for the return values.

Example 7:

```
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");
         exit(1);
    }
    return;
}
```

Calling in the lowest layer
2.3.3.2

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 example 8, which contains code to call the `nusers` service.

Example 8:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int addrlen, 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;
    addrlen = sizeof(struct sockaddr_in);
    bzero ((char*) &server_addr, sizeof (server_addr));
    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);
    }
    clnt_destroy(client);
    close(sock);
    exit(0)
    }
```

The following text explains the RPC portions of the client source code in example 8.

Lines 34 through 37: The client pointer is encoded with the transport mechanism. The `callrpc` routine uses UDP; thus, it calls `clntudp_create` to get a client pointer. The `clntudp_create` parameters are the server address, the program number, the version number, a time-out value (between tries), and a pointer to a socket. The final `clnt_call` argument (line 41) is the total time to wait for a response. Thus, the number of tries is the `clnt_call` time-out divided by the `clntudp_create` time-out.

To get TCP/IP and to make a stream connection, the call to `clntudp_create` is replaced with the following call to `clnttcp_create`:

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

There is no time-out argument; instead, you must specify the receive (`inputsize`) and send (`outputsize`) buffer sizes. When the `clnttcp_create` call is made, a TCP connection is established. All RPCs using that client handle use this connection. (On the server side of an RPC using TCP, `svcudp_create` is replaced by `svctcp_create`.)

Lines 41 through 42: The low-level version of `callrpc` is `clnt_call`. The `clnt_call` parameters are a client pointer (rather than a host name), 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 the destination for the return value, and the number of seconds to wait for a reply.

Line 47: The `clnt_destroy` call deallocates any space associated with the client handle, but it closes the socket associated with the client handle only if the RPC library opened it. If a user opened the socket, it stays open because, if multiple client handles are using the same socket, you can close one handle without destroying the socket that other handles are using.

The `clnt_create` interface greatly simplifies the method for accessing the low-level RPC features. Like `clnttcp_create` and `clntudp_create`, `clnt_create` returns a pointer to a client structure. However, `clnt_create` removes much of the work associated with the other two calls by allowing you to pass in the host name and protocol type as parameters of type character pointer (`char*`).

The syntax of the `clnt_create` call is as follows:

```
struct CLIENT *cp;

char *hostname;         /* hostname string */
unsigned int prog;      /* the program number */
unsigned int vers;      /* the version number */
char *protocol;         /* currently “udp” or “tcp” */

cp = clnt_create(hostname, prog, vers, protocol);
```

Using this interface, lines 22 through 35 of example 8 could be replaced by the following line:

```
if ((client = clnt_create(argv[1], RUSERSPROG, RUSERSVERS,“udp”)) == NULL)
{
```

If a TCP delivery mechanism were preferred, string `tcp` would replace string `udp` in this call.

If `clnt_create` fails, it returns the value `NULL`; the error can be identified with a call to `clnt_pcreateerror`. `clnt_create` can fail for the following reasons:

```
RPC_HOSTUNKNOWN;        /* host not known by the system */
RPC_SYSTEMERR;          /* host not in Internet Address Family */
RPC_UNKNOWNPROTO;       /* unknown protocol...not “udp” or “tcp” */
```

Select processing 2.3.3.3

Suppose a routine is processing RPC requests while performing another activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling `svc_run`. But if the other activity involves waiting on a file descriptor, the `svc_run` call will not work. Example 9 shows the code for `svc_run`.

Example 9:

```
void
svc_run()
{
    fd_set readfds;
    extern int errno;
    for (;;) {
    readfds = svc_fdset;
    switch (select(32, &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_getreq` (or `svc_getreqset`) yourself. To do so, you must know only the file descriptors of the sockets associated with the programs for which you are waiting. Thus, you can have your own `select`(2), which waits on both the RPC socket and your own descriptors.

> **Note:** `svc_fdset` is a global bit mask of all file descriptors that RPC is using for services. It can change any time an RPC library routine is called. Descriptors are constantly being opened and closed (for example, for TCP connections).

TCP processing
2.3.3.4

In example 10, the initiator of the `snd()` RPC takes its standard input and sends it to server `rcv()`, which prints it on standard output. The RPC uses TCP. This example also illustrates an XDR procedure that behaves differently on serialization than on deserialization.

Example 10:

```
/*
 * The xdr routine:
 *      on decode, read from the network, write to the file
 *      on encode, read from the file, write to the network
 *
 *      Returns 1 if successful
 *      Returns 0 if an xdr failure occurs
 *      Exits if a fread or fwrite fails.
 */

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

xdr_rcp(xdrs, fp)
XDR *xdrs;
FILE *fp;
{

        unsigned long size;
        char buf[BUFSIZ];
        char *p;

        if (xdrs->x_op == XDR_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”);
                        exit(1);
                        }
```

(continued)

```
                }

                p = buf;

                /* On ENCODE, this operation is a “write to network”
                 * On DECODE, this operation is a “read from network”
                */

                if (!xdr_bytes(xdrs, &p, &size, BUFSIZ)) {
                        return(0);              /* an XDR failure */
                }


                if (size == 0) {            /* Normal exit */
                              return(1);
                }

                if (xdrs->x_op == XDR_DECODE) {
                        if (fwrite(buf, sizeof(char), size, fp) != size) {
                                fprintf(stderr, “fwrite error\n”);
                                exit(1);
                         }
                }
        } /* end while */
}

/*
* The sender routines
 */

#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>

int callrpctcp();

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

        if (argc < 2) {
                fprintf(stderr, “usage: %s servername\n”,argv[0]);
                exit(1);
```

(continued)

```
        }

        if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP, RCPVERS,
                xdr_rcp, stdin, xdr_void, 0) != 0)) {
                clnt_perrno(err);
                fprintf(stderr, “can’t make the RPC call\n”);
                exit(1);
        }
}

callrpctcp(host, prognum, procnum, versnum, inproc, in, outproc, out)
char *host;
int prognum;
int procnum;
int versnum;
xdr_proc_t inproc;
char *in;
xdr_proc_t outproc;
char *out;
{
        struct sockaddr_in server_addr;
        int sock = RPC_ANYSOCK;
        enum clnt_stat client_stat;
        struct hostent *hp;
        register CLIENT *client;
        struct timeval total_timeout;

        if ((hp = gethostbyname(host)) == NULL) {
                fprintf(stderr, “can’t get address for ’%s’\n”,host);
                exit(1);
        }

        bzero((char*)&server_addr, sizeof(server_addr));
        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,
                &sock, BUFSIZ, BUFSIZ)) == NULL ) {

                perror(“rpctcp_create”);
                exit(1);
        }

        total_timeout.tv_sec  = 20;
        total_timeout.tv_usec = 0;
```

(continued)

```
        client_stat = clnt_call(client, procnum, inproc, in,
                outproc, out, total_timeout);
        clnt_destroy(client);
        return((int)client_stat);
}

/*
 * The receiving routines
 */

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

main()
{
       register SVCXPRT *transp;

       if ((transp = svctcp_create(RPC_ANYSOCK, BUFSIZ, BUFSIZ)) == NULL) {
                fprintf(stderr, “svctcp_create: error\n”);
                exit(1);
       }

       pmap_unset(RCPPROG, RCPPROC);      /* remove any old entry */

       if (!svc_register(transp, RCPPROG, RCPVERS,
                rcp_service, IPPROTO_TCP)) {

                fprintf(stderr, “svc_register: error\n”);
                exit(1);
       }
       svc_run();            /* should never return */
       fprintf(stderr, “svc_run should not return, but it did!\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)) {
                        fprintf(stderr, “err: rcp NULL service\n”);
                }
                return;
```

(continued)

```
        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 send reply\n");
                }
                return;

        default:
                svcerr_noproc(transp);
                return;

                }       /* end switch */
}
```

Callback processing
2.4

Occasionally, it is useful to have a server become a client and make an RPC back to the process that is its client. This is called *callback processing*. An example of its use is remote debugging, in which the client is a window system program and the server is a debugger running on the remote machine. Usually, the user clicks a mouse button at the debugging window, which brings up a debugger command and then makes an RPC 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 must make an RPC to the window program, informing the user that it has reached a breakpoint.

To do callback processing, you need a program number on which to make the RPC. Because this will be a dynamically generated program number, it should be in the transient range, 0x40000000 to 0x5fffffff. In example 11, the `gettransient` routine returns a valid program number in the transient range and registers it with the portmapper. It talks only to the portmapper that is running on the same machine as the `gettransient` routine itself. The call to `pmap_set` is a test and set operation; that is, it indivisibly tests whether a program number has already been registered, and, if it has not, reserves it. This prevents more than one process from reserving the same program number. On return, the `sockp` argument contains a socket that can be used as the argument to an `svcudp_create` or `svctcp_create` call.

Example 11:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
gettransient(proto, vers, sockp)
   int proto, vers, *sockp;
{
   static int prognum = 0x40000000;
   int s, len, socktype;
   struct sockaddr_in addr;
   switch(proto) {
       case IPPROTO_UDP:
           socktype = SOCK_DGRAM;
           break;
       case IPPROTO_TCP:
           socktype = SOCK_STREAM;
           break;
       default:
           fprintf(stderr, “unknown protocol type\n”);
           return 0;
   }
   if (*sockp == RPC_ANYSOCK) {
       if ((s = socket(AF_INET, socktype, 0)) <0) {
           perror(“socket”);
           return (0);
       }
       *sockp = s;
   }
   else
       s = *sockp;
   bzero ((char*) &addr, sizeof (addr));
   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, addr.sin_port))
       continue;
   return (prognum–1);
)
```

The two programs in example 12 illustrate how to use the `gettransient` routine. The client makes an RPC to the server, passing it a transient program number. The client then waits to receive a callback from the server at that program number. The server registers the program `EXAMPLEPROG`, so that it can receive the RPC 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, using the program number it received earlier.

Example 12:

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>
int callback();
char hostname[256];
main(argc, argv)
   int argc;
   char **argv;
{
   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 (ans != RPC_SUCCESS) {
       fprintf(stderr, "call: ");
       clnt_perrno(ans);
       fprintf(stderr, "\n");
       exit(1)
```

(continued)

```
   }
   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: rusersd\n”);
               exit(1);
           }
           exit(0);
       case 1:
           if (!svc_getargs(transp, xdr_void, 0)) {
               svcerr_decode(transp);
               exit(1);
           }
           fprintf(stderr, “client got callback\n”);
           if (!svc_sendreply(transp, xdr_void, 0)) {
               fprintf(stderr, “err: rusersd”);
               exit(1);
           }
   }
}
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>
char *getnewprog();
char hostname[256];
int docallback();
int pnum;        /* program number for callback routine */
main(argc, argv)
int argc
   char **argv;
{
```

(continued)

```
   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:\n”);
       clnt_perrno(ans);
       fprintf(stderr, ”\n”);
   }
}
```

Other uses of the RPC protocol
2.5

The RPC protocol is intended for use in calling remote procedures: each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which protocols other than RPC can be implemented. For example, you can use the RPC message protocol for batching (or pipelining) and broadcast RPC.

Batching 2.5.1

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

Batching allows a client to send an arbitrarily large sequence of call messages to a server; reliable byte stream protocols (such as TCP/IP) are used for transport. In the case of batching, the client never waits for a reply from the server, and the server does not send replies to batch requests. A nonbatched RPC command usually terminates a sequence of batch calls to flush the pipeline (with positive acknowledgment).

Because the server does not respond to every call, the client can generate new calls in parallel with the server's execution of previous calls. Furthermore, the TCP/IP implementation can buffer up many call messages and can send them to the server in one `write`(2) system call. This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes and the total elapsed time required for a series of calls.

Assume that a string-rendering service (such as a window system) has two similar calls: one renders a string and returns void results; the other renders a string and remains silent. The service (using the TCP/IP transport) might look like example 13.

Example 13:

```
/*
 * This is the file window.h
 */

#define WINDOWPROG (0x20100003)     /* PROGNUM within the USER range */
#define WINDOWVERS (1)

/* Windowing Procedures */

#define RENDERSTRING (1)
#define RENDERSTRING_BATCHED (2)

/* end of “window.h” */


/*
 * This is the file window_svc.c
 */

#include <stdio.h>
#include <rpc/rpc.h>
#include “window.h”

void windowdispatch();

main()
{
        SVCXPRT *transp;

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL) {
                fprintf(stderr, “can’t create the RPC server\n”);
                exit(1);
        }

        /* remove any old mapping that may be left over */

        pmap_unset(WINDOWPROG, WINDOWVERS);

        if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
                windowdispatch, IPPROTO_TCP)) {
```

(continued)

```
                fprintf(stderr, “can’t register WINDOW service\n”);
                exit(1);
        }

        svc_run();       /* never returns */

        fprintf(stderr, “svc_run should never return, but it did!\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 NULL RPC call\n”);
                }
                return;

        case RENDERSTRING:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, “can’t decode RENDERSTRING args\n”);

                /* tell the caller they made an error */

                        svcerr_decode(transp);
                        break;
                }

                /* Code here to actually render the string... */

                /* Now send reply to the caller...*/
```

(continued)

```
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, “can’t reply to RPC call\n”);
                        return;
                }
                        break;

        case RENDERSTRING_BATCHED:

                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, “can’t decode BATCHED args\n”);

                        /* since batched, silent in face of protocol errs */

                        break;
                }

                /* Code here to actually render the string... */

                /* Since batched, send NO reply to the caller...*/

                break;

        default:
                svcerr_noproc(transp);
                return;

        }       /* end switch */

    /* Free the string allocated when the arguments were decoded... */

        svc_freeargs(transp, xdr_wrapstring, &s);

}
```

The service could have one procedure that takes the string and a Boolean to indicate whether the procedure should respond.

For a client to take advantage of batching, the client must perform RPCs on a TCP-based transport, and the actual calls must have the following attributes:

- The XDR routine result must be 0 (`NULL`).
- The time-out of the RPC must be 0.

Example 14 shows a client that uses batching to render a series of strings; the batching is flushed when the client gets a null string.

Example 14:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include “window.h”
#include <sys/time.h>

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

        client = clnt_create(argv[1], WINDOWPROG, WINDOWVERS, “tcp”);
        if (client == NULL) {
                fprintf(stderr, “clnt_create [%s] failed\n”,argv[1]);
                exit(1);
        }

        total_timeout.tv_sec = 0;
        total_timeout.tv_usec = 0;

   /* Somewhat dangerous...the scanf() could overflow the buffer */

        while (scanf(”%s”, s) != EOF) {
                client_stat = clnt_call(client, RENDERSTRING_BATCHED,
                    xdr_wrapstring, &s, NULL, NULL, total_timeout);
                if (client_stat != RPC_SUCCESS) {
                        clnt_perror(client, “batched rpc”);
                        exit(-1);
                }
        }         /* end while */
        /* Now flush the pipeline */

        total_timeout.tv_sec = 20;
```

(continued)

```
        client_stat = clnt_call(client, NULLPROC,
              xdr_void, NULL, xdr_void, NULL, total_timeout);

        if (client_stat != RPC_SUCCESS) {
                clnt_perror(client, "rpc");
                exit(-1);

        }
        /* all done...now clean up */

        clnt_destroy(client);
}
```

Because the server sends no message, the clients cannot be notified of any failures that occur. Therefore, clients must handle errors on their own.

Example 14 was completed to render all 2000 lines in the `/etc/termcap` file. The rendering service did nothing but delete the lines. The example was run (by Sun Microsystems) in the following configurations with the following results:

Configuration	Results
Machine to itself, regular RPC	50 seconds
Machine to itself, batched RPC	16 seconds
Machine to another, regular RPC	52 seconds
Machine to another, batched RPC	10 seconds

Running `fscanf` (see `scanf`(3)) on file `/etc/termcap` requires only 6 seconds. These timings show the advantage of protocols that allow for overlapped execution, although these protocols are often difficult to design.

Broadcast RPC
2.5.2

In broadcast protocols based on RPC, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet-based protocols (such as UDP/IP) for transport. Servers that support broadcast protocols respond only when the request is processed successfully, and they are silent when errors occur.

The portmapper is a daemon that converts RPC program numbers into DARPA protocol port numbers (see `portmap`(8)). You cannot do broadcast RPC without the portmapper, `portmap`, in conjunction with standard RPC protocols. The following are the main differences between broadcast RPC and normal RPC:

- Normal RPC expects one answer; broadcast RPC expects many answers (one or more answers from each responding machine).
- Only packet-oriented (connectionless) transport protocols such as UDP/IP can support broadcast RPC.
- The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, 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 through the broadcast RPC mechanism.
- Broadcast request sizes are limited to the maximum transmission unit (mtu) of the local network.

The following is a synopsis of broadcast RPC:

```
#include <rpc/pmap_clnt.h>
enum clnt_stat clnt_stat;
clnt_stat =
clnt_broadcast(prog, vers, proc, xargs, argsp, xresults,
    resultsp, eachresult)
u_long     prog;         /* program number */
u_long     vers;         /* version number */
u_long     proc;         /* procedure number */
xdrproc_t    xargs;      /* xdr routine for args */
caddr_t     argsp;       /* pointer to args */
xdrproc_t    xresults;   /* xdr routine for results */
caddr_t     resultsp;    /* pointer to results */
bool_t (*eachresult)();  /* call with each result obtained*/
```

The `eachresult()` routine is called each time a valid result is obtained. It returns the following Boolean, which indicates whether the client 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, 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 return, the routine returns with RPC_TIMEDOUT. To interpret clnt_stat errors, feed the error code to clnt_perrno.