Remote Procedure Call
Programming Guide

Remote Procedure Call
Programming Guide

1. Introduction

This document is intended for programmers who wish to write network applications using remote procedure calls (explained below), thus avoiding low-level system primitives based on sockets. The reader must be familiar with the C programming language, and should have a working knowledge of network theory.

Programs that communicate over a network need a paradigm for communication. A low-level mechanism might send a signal on the arrival of incoming packets, causing a network signal handler to execute. A high-level mechanism would be the Ada `rendezvous`. The method used at Sun is the Remote Procedure Call (RPC) paradigm, in which a client communicates with a server. In this process, the client first calls a procedure 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 the procedure call returns to the client.

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

At the middle layer, the routines `registerrpc()` and `callrpc()` are used to make RPC calls: `registerrpc()` obtains a unique system-wide number, while `callrpc()` executes a remote procedure call. The `rnusers()` call is implemented using these two routines The middle-layer routines are designed for most common applications, and shield the user from knowing about sockets.

The lowest layer is used for more sophisticated applications, which may want to alter the defaults of the routines. At this layer, you can explicitly manipulate sockets used for transmitting RPC messages. This level should be avoided if possible.

Section 2 of this manual illustrates use of the highest two layers while Section 3 presents the low-level interface. Section 4 of the manual discusses miscellaneous topics. The final section summarizes all the entry points into the RPC system.

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

There is a diagram of the RPC paradigm on the next page.

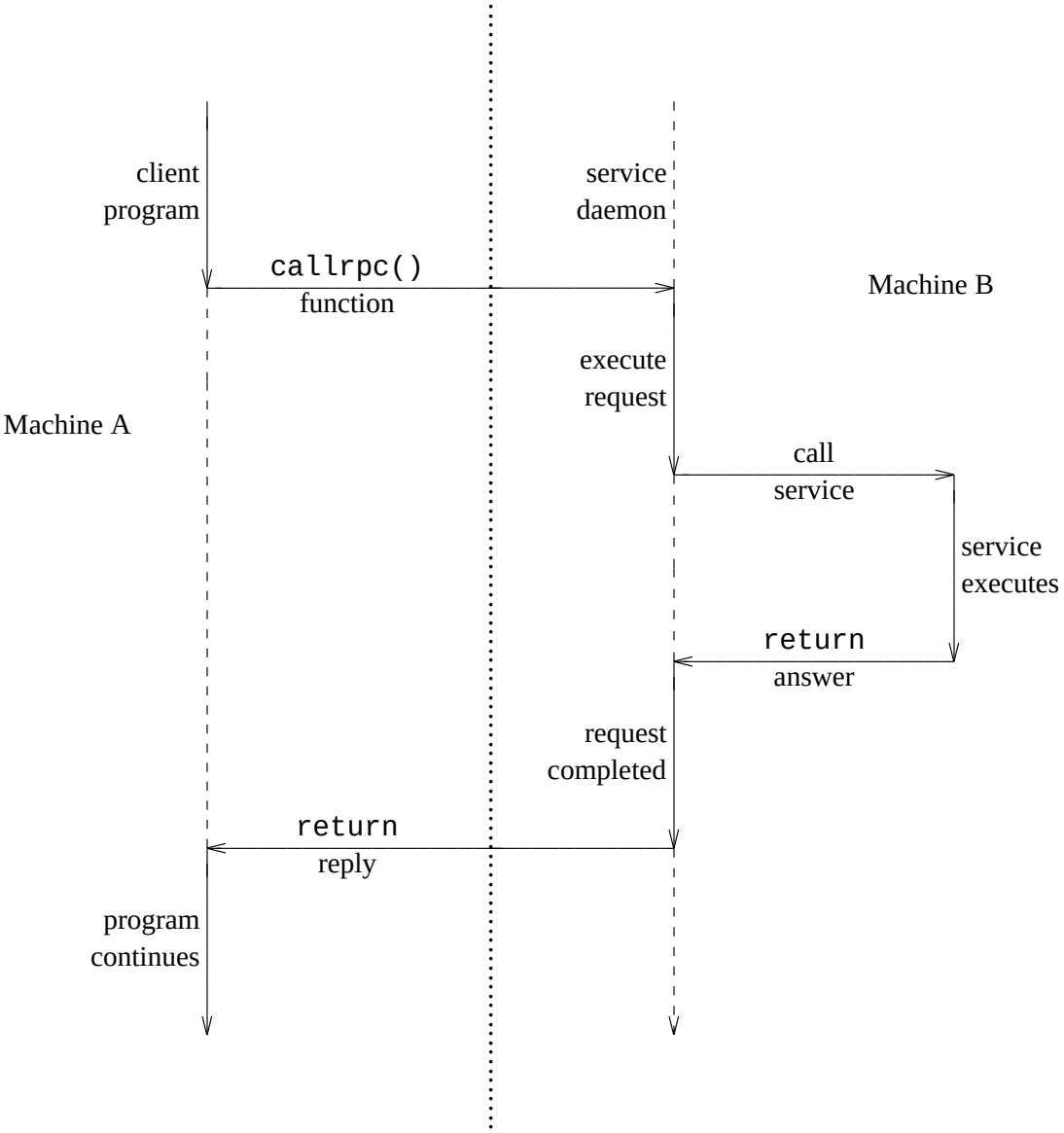

Figure 1: Network Communication with the Remote Procedure Call

2. Introductory Examples

2.1. Highest Layer

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

```
#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()` are included in the C library `libc.a`. Thus, the program above could be compiled with

```
% cc program.c
```

Some other library routines are `rstat()` to gather remote performance statistics, and `ypmatch()` to glean information from the yellow pages (YP). The YP library routines are documented on the manual page *ypclnt* (3N).

2.2. Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions `callrpc()` and `registerrpc()`. Using this method, another way to get the number of remote users is:

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

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

        if (argc < 2) {
                fprintf(stderr, "usage: nusers hostname\n");
                exit(-1);
        }
        if (callrpc(argv[1], RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
            xdr_void, 0, xdr_u_long, &nusers) != 0) {
                fprintf(stderr, "error: callrpc\n");
                exit(1);
        }
        printf("number of users on %s is %d\n", argv[1], nusers);
        exit(0);
}
```

A program number, version number, and procedure number defines each RPC procedure. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (adding a new procedure, for example), a new program number doesn't have to be assigned. When you want to call a procedure to find the number of remote users, you look up the appropriate program, version and procedure numbers in a manual, similar to when you look up the name of memory allocator when you want to allocate memory.

The simplest routine in the RPC library used to make remote procedure calls is `callrpc()`. It has eight parameters. The first is the name of the remote machine. The next three parameters are the program, version, and procedure numbers. The following two parameters define the argument of the RPC call, and the final two parameters are for the return value of the call. If it completes successfully, `callrpc()` returns zero, but nonzero otherwise. The exact meaning of the return codes is found in `<rpc/clnt.h>`, and is in fact an `enum clnt_stat` cast into an integer.

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` as its second return parameter, which is a pointer to where the long result will be placed. Since RUSERSPROC_NUM takes no argument, the argument parameter of `callrpc()` is `xdr_void`.

After trying several times to deliver a message, if `callrpc()` gets no answer, it returns with an error code. The delivery mechanism is UDP, which stands for User Datagram Protocol. 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 above might look like this:

```
char *
nuser(indata)
        char *indata;
{
        static int nusers;

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

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

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 would look like this:

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

char *nuser();

main()
{
        registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM, nuser,
            xdr_void, xdr_u_long);
        svc_run();              /* never returns */
        fprintf(stderr, "Error: svc_run returned!\n");
        exit(1);
}
```

The `registerrpc()` routine establishes what C procedure corresponds to each RPC procedure number. The first three parameters, RUSERPROG, RUSERSVERS, and RUSERSPROC_NUM are the program, version, and procedure numbers of the remote procedure to be registered; `nuser` is the name of the C procedure implementing it; and `xdr_void` and `xdr_u_long` are the types of the input to and output from the procedure.

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

Warning: the UDP transport mechanism can only deal with arguments and results less than 8K bytes in length.

2.3. Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 (536870912) according to the following chart:

Number			Assignment
0	-	1fffffff	defined by sun
20000000	-	3fffffff	defined by user
40000000	-	5fffffff	transient
60000000	-	7fffffff	reserved
80000000	-	9fffffff	reserved
a0000000	-	bfffffff	reserved
c0000000	-	dfffffff	reserved
e0000000	-	ffffffff	reserved

Sun Microsystems administers the first group of numbers, which should be identical for all Sun customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range. The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

The exact registration process for Sun defined numbers is yet to be established.

2.4. Passing Arbitrary Data Types

In the previous example, the RPC call passes a single `unsigned long`. RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions, by always converting them to a network standard called *eXternal Data Representation* (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called *serializing*, and the reverse process is called *deserializing*. The type field parameters of `callrpc()` and `registerrpc()` can be a built-in procedure like `xdr_u_long()` in the previous example, or a user supplied one. XDR has these built-in type routines:

```
xdr_int()       xdr_u_int()     xdr_enum()
xdr_long()      xdr_u_long()    xdr_bool()
xdr_short()     xdr_u_short()   xdr_string()
```

As an example of a user-defined type routine, if you wanted to send the structure

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

then you would call `callrpc` as

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

where `xdr_simple()` is written as:

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

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

An XDR routine returns nonzero (true in the sense of C) if it completes successfully, and zero otherwise. A complete description of XDR is in the *XDR Protocol Specification*, so this section only gives a few examples of XDR implementation.

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

```
xdr_array()       xdr_bytes()
xdr_reference()   xdr_union()
```

To send a variable array of integers, you might package them up as a structure like this

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

and make an RPC call such as

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

with `xdr_varintarr()` defined as:

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

This routine takes as parameters the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum allowable array size, the size of each array element, and an XDR routine for handling each array element.

If the size of the array is known in advance, then the following could also be used to send out an array of length SIZE:

```
int intarr[SIZE];

xdr_intarr(xdrsp, intarr)
        XDR *xdrsp;
        int intarr[];
{
        int i;

        for (i = 0; i < SIZE; i++) {
                if (!xdr_int(xdrsp, &intarr[i]))
                        return (0);
        }
        return (1);
}
```

XDR always converts quantities to 4-byte multiples when deserializing. Thus, if either of the examples above 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. It has four parameters, the same as 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 it gets the string length from `strlen()`, and on deserializing it creates a null-terminated string.

Here 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;
{
        int i;

        if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
                return (0);
        if (!xdr_reference(xdrsp, &finalp->simplep,
            sizeof(struct simple), xdr_simple);
                return (0);
        return (1);
}
```

3. Lower Layers of RPC

In the examples given so far, RPC takes care of many details automatically for you. In this section, we'll show you how you can change the defaults by using lower layers of the RPC library. It is assumed that you are familiar with sockets and the system calls for dealing with them. If not, consult *The IPC Tutorial* .

3.1. More on the Server Side

There are a number of assumptions built into `registerrpc()`. One is that you are using the UDP datagram protocol. Another is that you don't want to do anything unusual while deserializing, since the deserialization process happens automatically before the user's server routine is called. The server for the `nusers` program shown below is written using a lower layer of the RPC package, which does not make these assumptions.

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

int nuser();

main()
{
        SVCXPRT *transp;

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL){
                fprintf(stderr, "couldn't create an RPC server\n");
                exit(1);
        }
        pmap_unset(RUSERSPROG, RUSERSVERS);
        if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser,
            IPPROTO_UDP)) {
                fprintf(stderr, "couldn'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, "couldn't reply to RPC call\n");
                        exit(1);
                }
                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, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

First, the server gets a transport handle, which is used for sending out RPC messages. `registerrpc()` 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, the RPC library creates a socket on which to send out RPC calls. Otherwise, `svcudp_create()` expects its argument to be a valid socket number. If you specify your own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of `svcudp_create()` and `clntudp_create()` (the low-level client routine) must match.

When the user specifies RPC_ANYSOCK for a socket or gives an unbound socket, the system determines port numbers in the following way: when a server starts up, it advertises to a port mapper demon on its local machine, which picks a port number for the RPC procedure if the socket specified to `svcudp_create()` isn't already bound. When the `clntudp_create()` call is made with an unbound socket, the system queries the port mapper on the machine to which the call is being made, and gets the appropriate port number. If the port mapper is not running or has no port corresponding to the RPC call, the RPC call fails. Users can make RPC calls to the port mapper themselves. The appropriate procedure numbers are in the include file `<rpc/pmap_prot.h>`.

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 RUSERS from the port mapper's tables.

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

The user routine `nuser()` must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by `nuser()` that `registerrpc()` handles automatically. The first is that procedure NULLPROC (currently zero) returns with no arguments. This can be used as a simple test for detecting if a remote program is running. Second, there is a check for invalid procedure numbers. If one is detected, `svcerr_noproc()` is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via `svc_sendreply()`. Its first parameter is the SVCXPRT handle, the second is the XDR routine, and the third is a pointer to the data to be returned. Not illustrated above is how a server handles an RPC program that passes data. As an example, we can add a procedure RUSERSPROC_BOOL, which has an argument `nusers`, and returns TRUE or FALSE depending on whether there are nusers logged on. It would look like this:

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

        if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
                svcerr_decode(transp);
                return;
        }
        /*
         * code to set nusers = number of users
         */
        if (nuserquery == nusers)
                bool = TRUE;
        else
                bool = FALSE;
        if (!svc_sendreply(transp, xdr_bool, &bool){
                 fprintf(stderr, "couldn't reply to RPC call\n");
                 exit(1);
        }
        return;
}
```

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

3.2. Memory Allocation with XDR

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

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

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

It might be called from a server like this,

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

where `chararr` has already allocated space. If you want XDR to do the allocation, you would have to 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 call might look like this:

```
char *arrptr;

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

After using the character array, it can be freed with `svc_freeargs()`. In the routine `xdr_finalexample()` given earlier, if `finalp->string` was NULL in the call

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

then

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

frees the array allocated to hold `finalp->string`; otherwise, it 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 serializing 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 doesn't have to worry about the three modes. The XDR reference manual has examples of more sophisticated XDR routines that determine which of the three modes they are in to function correctly.

3.3. 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 the following code to call the `nusers` service:

```
#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, "cannot get addr for '%s'\n", argv[1]);
                exit(-1);
        }
        pertry_timeout.tv_sec = 3;
        pertry_timeout.tv_usec = 0;
        addrlen = sizeof(struct sockaddr_in);
        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) {
                perror("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); }
```

The low-level version of `callrpc()` is `clnt_call()`, which takes a CLIENT pointer rather than a host name. The parameters to `clnt_call()` are a CLIENT pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the

return value, a pointer to where the return value will be placed, and the time in seconds to wait for a reply.

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

The parameters to `clntudp_create()` are the server address, the length of the server address, the program number, the version number, a timeout value (between tries), and a pointer to a socket. The final argument to `clnt_call()` is the total time to wait for a response. Thus, the number of tries is the `clnt_call()` timeout divided by the `clntudp_create()` timeout.

There is one thing to note when using the `clnt_destroy()` call. It deallocates any space associated with the CLIENT handle, but it does not close the socket associated with it, which was passed as an argument to `clntudp_create()`. The reason is that if there are multiple client handles using the same socket, then it is possible to close one handle without destroying 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, &socket, inputsize,
        outputsize);
```

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

4. Other RPC Features

This section discusses some other aspects of RPC that are occasionally useful.

4.1. Select on the Server Side

Suppose a process is processing RPC requests while performing some other activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling `svc_run()`. But if the other activity involves waiting on a a file descriptor, the `svc_run()` call won't work. The code for `svc_run()` is as follows:

```
void
svc_run()
{
        int readfds;

        for (;;) {
                readfds = svc_fds;
                switch (select(32, &readfds, NULL, NULL, NULL)) {

                case -1:
                        if (errno == EINTR)
                                continue;
                        perror("rstat: select");
                        return;
                case 0:
                        break;
                default:
                        svc_getreq(readfds);
                }
        }
}
```

You can bypass `svc_run()` and call `svc_getreq()` yourself. All you need to know are the file descriptors of the socket(s) associated with the programs you are waiting on. Thus you can have your own `select()` that waits on both the RPC socket, and your own descriptors.

4.2. Broadcast RPC

The `pmap` and RPC protocols implement broadcast RPC. Here are the main differences between broadcast RPC and normal RPC calls:

1) Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).

2) Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UPD/IP.

3) 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.

4) 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.

4.2.1. Broadcast RPC Synopsis

```
#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 procedure `eachresult()` is called each time a valid result is obtained. It returns a boolean that indicates whether or not the client wants more responses.

```
bool_t                  done;

done =
eachresult(resultsp, raddr)
caddr_t                 resultsp;
struct sockaddr_in      *raddr;   /* address of machine that sent response */
```

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. To interpret `clnt_stat` errors, feed the error code to `clnt_perrno()`.

4.3. Batching

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

RPC messages can be placed in a ‘‘pipeline’’ of calls to a desired server; this is called batching. Batching assumes that: 1) each RPC call in the pipeline requires no response from the server, and the server does not send a response message; and 2) the pipeline of calls is transported on a reliable byte stream transport such as TCP/IP. Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Furthermore, the TCP/IP implementation can buffer up many call messages, and send them to the server in one `write` system call.

This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes, and the total elapsed time of a series of calls.

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

A contrived example of batching follows. 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:

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

void windowdispatch();

main()
{
        SVCXPRT *transp;

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL){
                fprintf(stderr, "couldn't create an RPC server\n");
                exit(1);
        }
        pmap_unset(WINDOWPROG, WINDOWVERS);
        if (!svc_register(transp, WINDOWPROG, WINDOWVERS, windowdispatch,
            IPPROTO_TCP)) {
                fprintf(stderr, "couldn'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, "couldn't reply to RPC call\n");
                        exit(1);
                }
```

```
        return;
        case RENDERSTRING:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, "couldn't decode arguments\n");
                        svcerr_decode(transp); /* tell caller he screwed up */
                        break;
                }
                /*
                 * call here to to render the string s
                 */
                if (!svc_sendreply(transp, xdr_void, NULL)) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                break;
        case RENDERSTRING_BATCHED:
                if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                        fprintf(stderr, "couldn't decode arguments\n");
                        /*
                         * we are silent in the face of protocol errors
                         */
                        break;
                }
                /*
                 * call here to to render the string s,
                 * but sends no reply!
                 */
                break;
        default:
                svcerr_noproc(transp);
                return;
        }
        /*
         * now free string allocated while decoding arguments
         */
        svc_freeargs(transp, xdr_wrapstring, &s);
}
```

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

In order for a client to take advantage of batching, the client must perform RPC calls on a TCP-based transport and the actual calls must have the following attributes: 1) the result's XDR routine must be zero (NULL), and 2) the RPC call's timeout must be zero.

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

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.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;
        char buf[1000];
        char *s = buf;
```

```
    /*
     * initial as in example 3.3
     */
    if ((client = clnttcp_create(&server_addr, WINDOWPROG,
        WINDOWVERS, &sock, 0, 0)) == NULL) {
            perror("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);
}
```

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 above example was completed to render all of the (2000) lines in the file */etc/termcap*. The rendering service did nothing but to throw the lines away. The example was run in the following four configurations: 1) machine to itself, regular RPC; 2) machine to itself, batched RPC; 3) machine to another, regular RPC; and 4) machine to another, batched RPC. The results are as follows: 1) 50 seconds; 2) 16 seconds; 3) 52 seconds; 4) 10 seconds. Running `fscanf()` on */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.

4.4. 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 filesystem, 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 used as a default is type *none*.

The authentication subsystem of the RPC package is open ended. That is, numerous types of authentication are easy to support. However, this section deals only with *unix* type authentication, which besides *none* is the only supported type.

4.4.1. The Client Side

When a caller creates a new RPC client handle as in:

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

the appropriate transport instance defaults the associate authentication handle to be

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

The RPC client can choose to use *unix* style authentication by setting `clnt->cl_auth` after creating the RPC client handle:

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

This causes each RPC call associated with `clnt` to carry with it the following authentication credentials structure:

```
/*
 * Unix style credentials.
 */
struct authunix_parms {
        u_long  aup_time;       /* credentials creation time */
        char    *aup_machname;  /* host name of where the client is calling */
        int     aup_uid;        /* client's UNIX effective uid */
        int     aup_gid;        /* client's current UNIX group id */
        u_int   aup_len;        /* the element length of aup_gids array */
        int     *aup_gids;      /* array of 4.2 groups to which user belongs */
};
```

These fields are set by `authunix_create_default()` by invoking the appropriate system calls.

Since the RPC user created this new style of authentication, he is responsible for destroying it with:

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

4.4.2. 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:

```
/*
 * An RPC Service request
 */
struct svc_req {
        u_long          rq_prog;        /* service program number */
        u_long          rq_vers;        /* service protocol version number*/
        u_long          rq_proc;        /* the desired procedure number*/
        struct opaque_auth rq_cred;     /* raw credentials from the "wire" */
        caddr_t         rq_clntcred;    /* read only, cooked credentials */
};
```

The `rq_cred` is mostly opaque, except for one field of interest: the style of authentication credentials:

```
/*
 * Authentication info.  Mostly opaque to the programmer.
 */
struct opaque_auth {
        enum_t  oa_flavor;      /* style of credentials */
        caddr_t oa_base;        /* address of more auth stuff */
        u_int   oa_length;      /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package guarantees the following to the service dispatch routine:

1) That the request's `rq_cred` is well formed. Thus the service implementor may inspect the request's `rq_cred.oa_flavor` to determine which style of authentication the caller used. The service implementor may also wish to inspect the other fields of `rq_cred` if the style is not one of the styles supported by the RPC package.

2) That the request's `rq_clntcred` field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. Remember that only *unix* style is currently supported, so (currently) `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is NULL, the service implementor may wish to inspect the other (opaque) fileds of `rq_cred` in case the service knows about a new type of authentication that the RPC package does not know about.

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

```
nuser(rqstp, tranp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        struct authunix_parms *unix_cred;
        int uid;
        unsigned long nusers;

        /*
         * we don't care about authentication for the null procedure
         */
        if (rqstp->rq_proc == NULLPROC) {
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(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 the caller is allow to call this procedure.
                 */
                if (uid == 16) {
                        svcerr_systemerr(transp);
                        return;
                }
                /*
                 * code here to compute the number of users
                 * and put in variable nusers
                 */
                if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                        fprintf(stderr, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

A few things should be noted here. First, it is customary not to check the authentication parameters associated with the NULLPROC (procedure number zero). Second, if the authentication parameter's type is not suitable for your service, you should call `svcerr_weakauth()`. And finally, the service protocol itself should return status for access denied; in the case of our example, the protocol does not have such a status, so we call the service primitive `svcerr_systemerr()` instead.

The last point underscores the relation between the RPC authentication package and the services; RPC deals only with authentication and not with individual services' access control. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

4.5. Using Inetd

An RPC server can be started from `inetd`. The only difference from the usual code is that `svcudp_create()` should be called as

```
transp = svcudp_create(0);
```

since `inet` passes a socket as file descriptor 0. Also, `svc_register()` should be called as

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

with the final flag as 0, since the program would already be registered by `inetd`. Remember that 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/servers for RPC services is

`rpc udp` *server program version*

where *server* is the C code implementing the server, and *program* and *version* are the program and version numbers of the service. The key word `udp` can be replaced by `tcp` for TCP-based RPC services.

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

```
rpc udp /usr/etc/rstatd 100001 1-2
```

5. More Examples

5.1. Versions

By convention, the first version number of program FOO is FOOVERS_ORIG and the most recent version is FOOVERS. Suppose there is a new version of the `user` program that returns an `unsigned short` rather than a `long`. If we name this version RUSERSVERS_SHORT, then a server that wants to support both versions would do a double register.

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG, nuser,
    IPPROTO_TCP)) {
        fprintf(stderr, "couldn't register RUSER service\n");
        exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT, nuser,
    IPPROTO_TCP)) {
        fprintf(stderr, "couldn't register RUSER service\n");
        exit(1);
}
```

Both versions can be handled by the same C procedure:

```
nuser(rqstp, tranp)
        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, "couldn't reply to RPC call\n");
                        exit(1);
                }
                return;
        case RUSERSPROC_NUM:
                /*
                 * code here to compute the number of users
                 * and put in variable nusers
                 */
                nusers2 = nusers;
                if (rqstp->rq_vers == RUSERSVERS_ORIG)
                        if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                                fprintf(stderr, "couldn't reply to RPC call\n");
                                exit(1);
                        }
                else
                        if (!svc_sendreply(transp, xdr_u_short, &nusers2) {
                                fprintf(stderr, "couldn't reply to RPC call\n");
                                exit(1);
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

5.2. TCP

Here is an example that is essentially `rcp`. 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/rpc.h>

xdr_rcp(xdrs, fp)
        XDR *xdrs;
        FILE *fp;
{
        unsigned long size;
        char buf[MAXCHUNK], *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), MAXCHUNK, fp))
                            == 0 && ferror(fp)) {
                                fprintf(stderr, "couldn't fread\n");
                                exit(1);
                        }
                }
                p = buf;
                if (!xdr_bytes(xdrs, &p, &size, MAXCHUNK))
                        return 0;
                if (size == 0)
                        return 1;
                if (xdrs->x_op == XDR_DECODE) {
                        if (fwrite(buf, sizeof(char), size, fp) != size) {
                                fprintf(stderr, "couldn't fwrite\n");
                                exit(1);
                        }
                }
        }
}
```

```
/*
 * The sender routines
 */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>

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

        if (argc < 2) {
                fprintf(stderr, "usage: %s server-name\n", argv[0]);
                exit(-1);
        }
        if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP, RCPVERS,
            xdr_rcp, stdin, xdr_void, 0) != 0)) {
                clnt_perrno(err);
                fprintf(stderr, " couldn't make RPC call\n");
                exit(1);
        }
}
```

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

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

```
/*
 * The receiving routines
 */
#include <stdio.h>
#include <rpc/rpc.h>

main()
{
        register SVCXPRT *transp;

        if ((transp = svctcp_create(RPC_ANYSOCK, 1024, 1024)) == 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");
                        exit(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;
                }
                exit(0);
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

5.3. Callback Procedures

Occasionally, it is useful to have a server become a client, and make an RPC call back the process which 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.

In order to do an RPC callback, you need a program number to make the RPC call on. Since this will be 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, in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it. On return, the `sockp` argument will contain a socket that can be used as the argument to an `svcudp_create()` or `svctcp_create()` call.

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

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

        switch(proto) {
                case IPPROTO_UDP:
                        socktype = SOCK_DGRAM;
                        break;
                case IPPROTO_TCP:
                        socktype = SOCK_STREAM;
                        break;
                default:
                        fprintf(stderr, "unknown protocol type\n");
                        return 0;
        }
        if (*sockp == RPC_ANYSOCK) {
                if ((s = socket(AF_INET, socktype, 0)) < 0) {
                        perror("socket");
                        return (0);
                }
                *sockp = s;
        }
        else
                s = *sockp;
        addr.sin_addr.s_addr = 0;
        addr.sin_family = AF_INET;
        addr.sin_port = 0;
        len = sizeof(addr);
        /*
         * may be already bound, so don't check for err
         */
        bind(s, &addr, len);
        if (getsockname(s, &addr, &len)< 0) {
                perror("getsockname");
                return (0);
        }
        while (pmap_set(prognum++, vers, proto, addr.sin_port) == 0)
                continue;
        return (prognum-1);
}
```

The following pair of 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 EXAMPELPROG, 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
 */
#include <stdio.h>
#include <rpc/rpc.h>

int callback();
char hostname[256];

main(argc, argv)
        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);
        }
        (void)svc_register(xprt, x, 1, callback, 0);

        ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEPROC_CALLBACK,
            EXAMPLEVERS, xdr_int, &x, xdr_void, 0);
        if (ans != 0) {
                fprintf(stderr, "call: ");
                clnt_perrno(ans);
                fprintf(stderr, "\n");
        }
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't have returned\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)  == FALSE) {
                                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)  == FALSE) {
                                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)
        char **argv;
{
        gethostname(hostname, sizeof(hostname));
        registerrpc(EXAMPLEPROG, EXAMPLEPROC_CALLBACK, EXAMPLEVERS,
            getnewprog, xdr_int, xdr_void);
        fprintf(stderr, "server going into svc_run\n");
        alarm(10);
        signal(SIGALRM, docallback);
        svc_run();
        fprintf(stderr, "Error: svc_run shouldn't have returned\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");
        }
}
```

Appendix A: Synopsis of RPC Routines

auth_destroy()

```
void
auth_destroy(auth)
        AUTH *auth;
```

A macro that destroys the authentication information associated with `auth`. Destruction usually involves deallocation of private data structures. The use of `auth` is undefined after calling `auth_destroy()`.

authnone_create()

```
AUTH *
authnone_create()
```

Creates and returns an RPC authentication handle that passes no usable authentication information with each remote procedure call.

authunix_create()

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
        char *host;
        int uid, gid, len, *aup_gids;
```

Creates and returns an RPC authentication handle that contains UNIX authentication information. The parameter `host` is the name of the machine on which the information was created; `uid` is the user's user ID; `gid` is the user's current group ID; `len` and `aup_gids` refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

authunix_create_default()

```
AUTH *
authunix_create_default()
```

Calls `authunix_create()` with the appropriate parameters.

callrpc()

```
callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
        char *host;
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
```

Calls the remote procedure associated with `prognum`, `versnum`, and `procnum` on the machine, `host`. The parameter `in` is the address of the procedure's argument(s), and `out` is the address of where to place the result(s); `inproc` is used to encode the procedure's parameters, and `outproc` is used to decode the procedure's results. This routine returns zero if it succeeds, or the value of `enum clnt_stat` cast to an integer if it fails. The routine `clnt_perrno()` is handy for translating failure statuses into messages. Warning: calling remote procedures with this routine uses UDP/IP as a transport; see `clntudp_create()` for restrictions.

clnt_broadcast()

```
enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
        resultproc_t eachresult;
```

Like `callrpc()`, except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls `eachresult`, whose form is

```
eachresult(out, addr)
        char *out;
        struct sockaddr_in *addr;
```

where `out` is the same as `out` passed to `clnt_broadcast()`, except that the remote procedure's output is decoded there; `addr` points to the address of the machine that sent the results. If `eachresult()` returns zero, `clnt_broadcast()` waits for more replies; otherwise it returns with appropriate status.

clnt_call()

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
        CLIENT *clnt; long procnum;
        xdrproc_t inproc, outproc;
        char *in, *out;
        struct timeval tout;
```

A macro that calls the remote procedure `procnum` associated with the client handle, `clnt`, which is obtained with an RPC client creation routine such as `clntudp_create`. The parameter `in` is the address of the procedure's argument(s), and `out` is the address of where to place the result(s); `inproc` is used to encode the procedure's parameters, and `outproc` is used to decode the procedure's results; `tout` is the time allowed for results to come back.

clnt_destroy()

```
clnt_destroy(clnt)
        CLIENT *clnt;
```

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including `clnt` itself. Use of `clnt` is undefined after calling `clnt_destroy()`. Warning: client destruction routines do not close sockets associated with `clnt`; this is the responsibility of the user.

clnt_freeres()

```
clnt_freeres(clnt, outproc, out)
        CLIENT *clnt;
        xdrproc_t outproc;
        char *out;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter `out` is the address of the results, and `outproc` is the XDR routine describing the results in simple primitives. This routine returns one if the results were successfully freed, and zero otherwise.

clnt_geterr()

```
void
clnt_geterr(clnt, errp)
        CLIENT *clnt;
        struct rpc_err *errp;
```

A macro that copies the error structure out of the client handle to the structure at address `errp`.

clnt_pcreateerror()

```
void
clnt_pcreateerror(s)
        char *s;
```

Prints a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string `s` and a colon.

clnt_perrno()

```
void
clnt_perrno(stat)
        enum clnt_stat;
```

Prints a message to standard error corresponding to the condition indicated by `stat`.

clnt_perror()

```
clnt_perror(clnt, s)
        CLIENT *clnt;
        char *s;
```

Prints a message to standard error indicating why an RPC call failed; `clnt` is the handle used to do the call. The message is prepended with string `s` and a colon.

clntraw_create()

```
CLIENT *
clntraw_create(prognum, versnum)
        u_long prognum, versnum;
```

This routine creates a toy RPC client for the remote program `prognum`, version `versnum`. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see `svcraw_create()`. This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

clnttcp_create()

```
CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
        struct sockaddr_in *addr;
        u_long prognum, versnum;
        int *sockp;
        u_int sendsz, recvsz;
```

This routine creates an RPC client for the remote program `prognum`, version `versnum`; the client uses TCP/IP as a transport. The remote program is located at Internet address `*addr`. If `addr->sin_port` is zero, then it is set to the actual port that the remote program is listening on (the remote *portmap* service is consulted for this information). The parameter `*sockp` is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets `*sockp`. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters `sendsz` and `recvsz`; values of zero choose suitable defaults. This routine returns NULL if it fails.

clntudp_create()

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
        struct sockaddr_in *addr;
        u_long prognum, versnum;
        struct timeval wait;
        int *sockp;
```

This routine creates an RPC client for the remote program `prognum`, version `versnum`; the client uses use UDP/IP as a transport. The remote program is located at Internet address `*addr`. If `addr->sin_port` is zero, then it is set to actual port that the remote program is listening on (the remote *portmap* service is consulted for this information). The parameter `*sockp` is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets `*sockp`. The UDP transport resends the call message in intervals of `wait` time until a response is received or until the call times out. Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

get_myaddress()

```
void
get_myaddress(addr)
        struct sockaddr_in *addr;
```

Stuffs the machine's IP address into `*addr`, without consulting the library routines that deal with */etc/hosts*. The port number is always set to `htons(PMAPPORT)`.

pmap_getmaps()

```
struct pmaplist *
pmap_getmaps(addr)
        struct sockaddr_in *addr;
```

A user interface to the *portmap* service, which returns a list of the current RPC program-to-port mappings on the host located at IP address `*addr`. This routine can return NULL. The command `rpcinfo -p` uses this routine.

pmap_getport()

```
u_short
pmap_getport(addr, prognum, versnum, protocol)
        struct sockaddr_in *addr;
        u_long prognum, versnum, protocol;
```

A user interface to the *portmap* service, which returns the port number on which waits a service that supports program number `prognum`, version `versnum`, and speaks the transport protocol associated with protocol. A return value of zero means that the mapping does not exist or that the RPC system failured to contact the remote *portmap* service. In the latter case, the global variable `rpc_createerr` contains the RPC status.

pmap_rmtcall()

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum,
             inproc, in, outproc, out, tout, portp)
        struct sockaddr_in *addr;
        u_long prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
        struct timeval tout;
        u_long *portp;
```

A user interface to the *portmap* service, which instructs *portmap* on the host at IP address `*addr` to make an RPC call on your behalf to a procedure on that host. The parameter `*portp` will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in `callrpc()` and `clnt_call()`; see also `clnt_broadcast()`.

pmap_set()

```
pmap_set(prognum, versnum, protocol, port)
        u_long prognum, versnum, protocol;
        u_short port;
```

A user interface to the *portmap* service, which establishes a mapping between the triple `[prognum,versnum,protocol]` and `port` on the machine's *portmap* service. The value of protocol is most likely IPPROTO_UDP or IPPROTO_TCP. This routine returns one if it succeeds, zero otherwise.

pmap_unset()

```
pmap_unset(prognum, versnum)
        u_long prognum, versnum;
```

A user interface to the *portmap* service, which destroys all mappings between the triple `[prognum,versnum,*]` and `ports` on the machine's *portmap* service. This routine returns one if it succeeds, zero otherwise.

registerrpc()

```
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
        u_long prognum, versnum, procnum;
        char *(*procname)();
        xdrproc_t inproc, outproc;
```

Registers procedure `procname` with the RPC service package. If a request arrives for program `prognum`, version `versnum`, and procedure `procnum`, `procname` is called with a pointer to its parameter(s); `progname` should return a pointer to its static result(s); `inproc` is used to decode the parameters while `outproc` is used to encode the results. This routine returns zero if the registration succeeded, –1 otherwise.

Warning: remote procedures registered in this form are accessed using the UDP/IP transport; see `svcudp_create()` for restrictions.

rpc_createerr

```
struct rpc_createerr    rpc_createerr;
```

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine `clnt_pcreateerror()` to print the reason why.

svc_destroy()

```
svc_destroy(xprt)
        SVCXPRT *xprt;
```

A macro that destroys the RPC service transport handle, `xprt`. Destruction usually involves deallocation of private data structures, including `xprt` itself. Use of `xprt` is undefined after calling this routine.

svc_fds

```
int     svc_fds;
```

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the `select` system call. This is only of interest if a service implementor does not call `svc_run()`, but rather does his own asynchronous event processing. This variable is read-only (do not pass its address to `select`!), yet it may change after calls to `svc_getreq()` or any creation routines.

svc_freeargs()

```
svc_freeargs(xprt, inproc, in)
        SVCXPRT *xprt;
        xdrproc_t inproc;
        char *in;
```

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using `svc_getargs()`. This routine returns one if the results were successfully freed, and zero otherwise.

svc_getargs()

```
svc_getargs(xprt, inproc, in)
        SVCXPRT *xprt;
        xdrproc_t inproc;
        char *in;
```

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle, `xprt`. The parameter `in` is the address where the arguments will be placed; `inproc` is the XDR routine used to decode the arguments. This routine returns one if decoding succeeds, and zero otherwise.

svc_getcaller()

```
struct sockaddr_in
svc_getcaller(xprt)
        SVCXPRT *xprt;
```

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle, `xprt`.

svc_getreq()

```
svc_getreq(rdfds)
        int rdfds;
```

This routine is only of interest if a service implementor does not call `svc_run()`, but instead implements custom asynchronous event processing. It is called when the `select` system call has determined that an RPC request has arrived on some RPC socket(s); `rdfds` is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of `rdfds` have been serviced.

svc_register()

```
svc_register(xprt, prognum, versnum, dispatch, protocol)
        SVCXPRT *xprt;
        u_long prognum, versnum;
        void (*dispatch)();
        u_long protocol;
```

Associates `prognum` and `versnum` with the service dispatch procedure, `dispatch`. If `protocol` is non-zero, then a mapping of the triple `[prognum,versnum,protocol]` to `xprt->xp_port` is also established with the local *portmap* service (generally `protocol` is zero, IPPROTO_UDP or IPPROTO_TCP). The procedure `dispatch()` has the following form:

```
dispatch(request, xprt)
        struct svc_req *request;
        SVCXPRT *xprt;
```

The `svc_register` routine returns one if it succeeds, and zero otherwise.

svc_run()

```
svc_run()
```

This routine never returns. It waits for RPC requests to arrive and calls the appropriate service procedure (using `svc_getreq`) when one arrives. This procedure is usually waiting for a `select` system call to return.

svc_sendreply()

```
svc_sendreply(xprt, outproc, out)
        SVCXPRT *xprt;
        xdrproc_t outproc;
        char *out;
```

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter `xprt` is the caller's associated transport handle; `outproc` is the XDR routine which is used to encode the results; and `out` is the address of the results. This routine returns one if it succeeds, zero otherwise.

svc_unregister()

```
void
svc_unregister(prognum, versnum)
        u_long prognum, versnum;
```

Removes all mapping of the double `[prognum,versnum]` to dispatch routines, and of the triple `[prognum,versnum,*]` to port number.

svcerr_auth()

```
void
svcerr_auth(xprt, why)
        SVCXPRT *xprt;
        enum auth_stat why;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

svcerr_decode()

```
void
svcerr_decode(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that can't successfully decode its parameters. See also `svc_getargs()`.

svcerr_noproc()

```
void
svcerr_noproc(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that doesn't implement the desired procedure number the caller request.

svcerr_noprog()

```
void
svcerr_noprog(xprt)
        SVCXPRT *xprt;
```

Called when the desired program is not registered with the RPC package. Service implementors usually don't need this routine.

svcerr_progvers()

```
void
svcerr_progvers(xprt)
        SVCXPRT *xprt;
```

Called when the desired version of a program is not registered with the RPC package. Service implementors usually don't need this routine.

svcerr_systemerr()

```
void
svcerr_systemerr(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

svcerr_weakauth()

```
void
svcerr_weakauth(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls `svcerr_auth(xprt,AUTH_TOOWEAK)`.

svcraw_create()

```
SVCXPRT *
svcraw_create()
```

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see `clntraw_create()`. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

svctcp_create()

```
SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
        int sock;
        u_int send_buf_size, recv_buf_size;
```

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket `sock`, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, `xprt->xp_sock` is the transport's socket number, and `xprt->xp_port` is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of the `send` and `receive` buffers; values of zero choose suitable defaults.

svcudp_create()

```
SVCXPRT *
svcudp_create(sock)
        int sock;
```

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket `sock`, which may be RPC_ANYSOCK, in which case a new socket is created. If the socket is not bound to a local UDP port, then this routine binds it to an arbitrary port. Upon completion, `xprt->xp_sock` is the transport's socket number, and `xprt->xp_port` is the transport's port number. This routine returns NULL if it fails. Warning: since UDP-based RPC messages can only hold up to 8 Kbytes of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

xdr_accepted_reply()

```
xdr_accepted_reply(xdrs, ar)
        XDR *xdrs;
        struct accepted_reply *ar;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        char **arrp;
        u_int *sizep, maxsize, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter `arrp` is the address of the pointer to the array, while `sizep` is the address of the element count of the array; this element count cannot exceed `maxsize`. The parameter `elsize` is the `sizeof()` each of the array's elements, and `elproc` is an XDR filter that translates between the array elements' C form, and their external representation. This routine returns one if it succeeds, zero otherwise.

xdr_authunix_parms()

```
xdr_authunix_parms(xdrs, aupp)
        XDR *xdrs;
        struct authunix_parms *aupp;
```

Used for describing UNIX credentials, externally. This routine is useful for users who wish to generate these credentials without using the RPC authentication package.

xdr_bool()

```
xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either one or zero. This routine returns one if it succeeds, zero otherwise.

xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
        char **sp;
        u_int *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter `sp` is the address of the string pointer. The length of the string is located at address `sizep`; strings cannot be longer than `maxsize`. This routine returns one if it succeeds, zero otherwise.

xdr_callhdr()

```
void
xdr_callhdr(xdrs, chdr)
        XDR *xdrs;
        struct rpc_msg *chdr;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_callmsg()

```
xdr_callmsg(xdrs, cmsg)
        XDR *xdrs;
        struct rpc_msg *cmsg;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_double()

```
xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

A filter primitive that translates between C `double` precision numbers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C `enum`s (actually integers) and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C `float`s and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_inline()

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
        int len;
```

A macro that invokes the in-line routine associated with the XDR stream, `xdrs`. The routine returns a pointer to a contiguous piece of the stream's buffer; `len` is the byte length of the desired buffer. Note that pointer is cast to `long *`. Warning: `xdr_inline()` may return 0 (NULL) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int()

```
xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_long()

```
xdr_long(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

A filter primitive that translates between C `long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        u_int cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter `cp` is the address of the opaque object, and `cnt` is its size in bytes. This routine returns one if it succeeds, zero otherwise.

xdr_opaque_auth()

```
xdr_opaque_auth(xdrs, ap)
        XDR *xdrs;
        struct opaque_auth *ap;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_pmap()

```
xdr_pmap(xdrs, regs)
        XDR *xdrs;
        struct pmap *regs;
```

Used for describing parameters to various *portmap* procedures, externally. This routine is useful for users who wish to generate these parameters without using the `pmap` interface.

xdr_pmaplist()

```
xdr_pmaplist(xdrs, rp)
        XDR *xdrs;
        struct pmaplist **rp;
```

Used for describing a list of port mappings, externally. This routine is useful for users who wish to generate these parameters without using the `pmap` interface.

xdr_reference()

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

A primitive that provides pointer chasing within structures. The parameter `pp` is the address of the pointer; `size` is the `sizeof()` the structure that `*pp` points to; and `proc` is an XDR procedure that filters the structure between its C form and its external representation. This routine returns one if it succeeds, zero otherwise.

xdr_rejected_reply()

```
xdr_rejected_reply(xdrs, rr)
        XDR *xdrs;
        struct rejected_reply *rr;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC-style messages without using the RPC package.

xdr_replymsg()

```
xdr_replymsg(xdrs, rmsg)
        XDR *xdrs;
        struct rpc_msg *rmsg;
```

Used for describing RPC messages, externally. This routine is useful for users who wish to generate RPC style messages without using the RPC package.

xdr_short()

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C `short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_string()

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        u_int maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot cannot be longer than `maxsize`. Note that `sp` is the address of the string's pointer. This routine returns one if it succeeds, zero otherwise.

xdr_u_int()

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C `unsigned` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_long()

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C `unsigned long` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_u_short()

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C `unsigned short` integers and their external representations. This routine returns one if it succeeds, zero otherwise.

xdr_union()

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C `union` and its corresponding external representation. The parameter `dscmp` is the address of the union's discriminant, while `unp` in the address of the union. This routine returns one if it succeeds, zero otherwise.

xdr_void()

```
xdr_void()
```

This routine always returns one.

xdr_wrapstring()

```
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
        char **sp;
```

A primitive that calls `xdr_string(xdrs,sp,MAXUNSIGNED);` where MAXUNSIGNED is the maximum value of an unsigned integer. This is handy because the RPC package passes only two parameters XDR routines, whereas `xdr_string()`, one of the most frequently used primitives, requires three parameters. This routine returns one if it succeeds, zero otherwise.

xprt_register()

```
void
xprt_register(xprt)
        SVCXPRT *xprt;
```

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable `svc_fds`. Service implementors usually don't need this routine.

xprt_unregister()

```
void
xprt_unregister(xprt)
        SVCXPRT *xprt;
```

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable `svc_fds`. Service implementors usually don't need this routine.