Remote Procedure Call Programming Guide

This document assumes a working knowledge of network theory. It is intended for programmers who wish to write network applications using remote procedure calls (explained below), and who want to understand the RPC mechanisms usually hidden by the *rpcgen(1)* protocol compiler. *rpcgen* is described in detail in the previous chapter, the **rpcgen** *Programming Guide*.

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

What are remote procedure calls? Simply put, they are the high-level communications paradigm used in the operating system. RPC presumes the existence of low-level networking mechanisms (such as TCP/IP and UDP/IP), and upon them it implements a logical client to server communications system designed specifically for the support of network applications. With RPC, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

1. Layers of RPC

The RPC interface can be seen as being divided into three layers.[1]

The Highest Layer: The highest layer is totally transparent to the operating system, machine and network upon which is is run. It's probably best to think of this level as a way of *using* RPC, rather than as a *part of* RPC proper. Programmers who write RPC routines should (almost) always make this layer available to others by way of a simple C front end that entirely hides the networking.

To illustrate, at this level a program can simply make a call to *rnusers*(), a C routine which returns the number of users on a remote machine. The user is not explicitly aware of using RPC — they simply call a procedure, just as they would call *malloc()*.

The Middle Layer: The middle layer is really "RPC proper." Here, the user doesn't need to consider details about sockets, the UNIX system, or other low-level implementation mechanisms. They simply make remote procedure calls to routines on other machines. The selling point here is simplicity. It's this layer that allows RPC to pass the "hello world" test — simple things should be simple. The middle-layer routines are used for most applications.

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

The middle layer is unfortunately rarely used in serious programming due to its inflexibility (simplicity). It does not allow timeout specifications or the choice of transport. It allows no UNIX process control or flexibility in case of errors. It doesn't support multiple kinds of call authentication. The programmer rarely needs all these kinds of control, but one or two of them is often necessary.

The Lowest Layer: The lowest layer does allow these details to be controlled by the programmer, and for that reason it is often necessary. Programs written at this level are also most efficient, but this is rarely a real issue — since RPC clients and servers rarely generate heavy network loads.

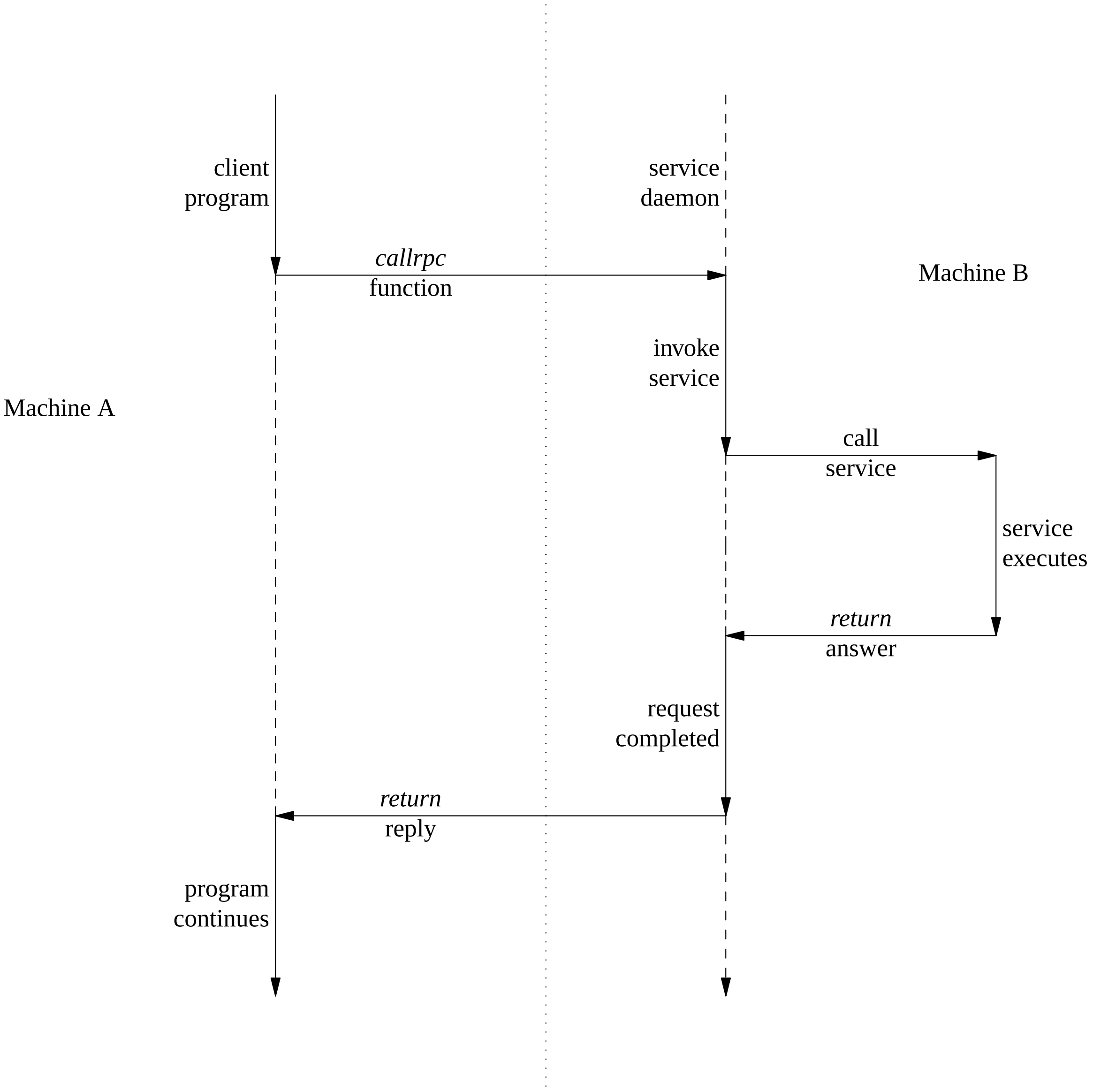
Although this document only discusses the interface to C, remote procedure calls can be made from any language. Even though this document discusses RPC when it is used to communicate between processes on different machines, it works just as well for communication between different processes on the same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

[1] For a complete specification of the routines in the remote procedure call Library, see the *rpc(3N)* manual page.

1.1. The RPC Paradigm

Here is a diagram of the RPC paradigm:

Figure 1-1 *Network Communication with the Remote Reocedure Call*



2. Higher Layers of RPC

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 RPC library routine *rnusers()* as illustrated below:

```
#include <stdio.h>

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

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

RPC library routines such as *rnusers()* are in the RPC services library *librpcsvc.a* Thus, the program above should be compiled with

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

rnusers(), like the other RPC library routines, is documented in section 3R of the *System Interface Manual for the Sun Workstation*, the same section which documents the standard Sun RPC services. See the *intro(3R)* manual page for an explanation of the documentation strategy for these services and their RPC protocols.

Here are some of the RPC service library routines available to the C programmer:

Table 3-3 *RPC Service Library Routines.TS*

Routine	*Description*
rnusers	*Return number of users on remote machine*
rusers	*Return information about users on remote machine*
havedisk	*Determine if remote machine has disk*
rstats	*Get performance data from remote kernel*
rwall	*Write to specified remote machines*
yppasswd	*Update user password in Yellow Pages*

Other RPC services — for example *ether() mount rquota()* and *spray* — are not available to the C programmer as library routines. They do, however, have RPC program numbers so they can be invoked with *callrpc()* which will be discussed in the next section. Most of them also have compilable *rpcgen(1)* protocol description files. (The *rpcgen* protocol compiler radically simplifies the process of developing network applications. See the **rpcgen** *Programming Guide* for detailed information about *rpcgen* and *rpcgen* protocol description files).

2.2. Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions *callrpc()* and *registerrpc()* Using this method, the number of remote users can be gotten as follows:

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

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

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

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

The simplest way of making remote procedure calls is with the the RPC library routine *callrpc()* It has eight parameters. The first is the name of the remote server machine. The next three parameters are the program, version, and procedure numbers—together they identify the procedure to be called. The fifth and sixth parameters are an XDR filter and an argument to be encoded and passed to the remote procedure. The final two parameters are a filter for decoding the results returned by the remote procedure and a pointer to the place where the procedure's results are to be stored. Multiple arguments and results are handled by embedding them in structures. If *callrpc()* completes successfully, it returns zero; else it returns a nonzero value. The return codes (of type cast into an integer) are found in *<rpc/clnt.h>*.

Since data types may be represented differently on different machines, *callrpc()* needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). 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;
{
    unsigned long 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 <rpc/rpc.h>
#include <utmp.h>
#include <rpcsvc/rusers.h>

char *nuser();

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

The *registerrpc()* routine registers a C procedure as corresponding to a given RPC procedure number. The first three parameters, *RUSERPROG*, *RUSERSVERS*, and *RUSERSPROC_NUM* are the program, version, and procedure numbers of the remote procedure to be registered; *nuser()* is the name of the local procedure that implements the remote procedure; and *xdr_void()* and *xdr_u_long()* are the XDR filters for the remote procedure's arguments and results, respectively. (Multiple arguments or multiple results are passed as structures).

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

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

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

2.3. Assigning Program Numbers

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

`0x0 - 0x1fffffff`	Defined by Sun
`0x20000000 - 0x3fffffff`	Defined by user
`0x40000000 - 0x5fffffff`	Transient
`0x60000000 - 0x7fffffff`	Reserved
`0x80000000 - 0x9fffffff`	Reserved
`0xa0000000 - 0xbfffffff`	Reserved
`0xc0000000 - 0xdfffffff`	Reserved
`0xe0000000 - 0xffffffff`	Reserved

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

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

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

Please include a compilable *rpcgen* “.x” file describing your protocol. You will be given a unique program number in return.

The RPC program numbers and protocol specifications of standard Sun RPC services can be found in the include files in */usr/include/rpcsvc*. These services, however, constitute only a small subset of those which have been registered. The complete list of registered programs, as of the time when this manual was printed, is:

Table 3-2 *RPC Registered Programs*

RPC Number	***Program***	***Description***
100000	PMAPPROG	*portmapper*
100001	RSTATPROG	*remote stats*
100002	RUSERSPROG	*remote users*
100003	NFSPROG	*nfs*
100004	YPPROG	*Yellow Pages*
100005	MOUNTPROG	*mount demon*
100006	DBXPROG	*remote dbx*
100007	YPBINDPROG	*yp binder*
100008	WALLPROG	*shutdown msg*
100009	YPPASSWDPROG	*yppasswd server*
100010	ETHERSTATPROG	*ether stats*
100011	RQUOTAPROG	*disk quotas*
100012	SPRAYPROG	*spray packets*
100013	IBM3270PROG	*3270 mapper*
100014	IBMRJEPROG	*RJE mapper*
100015	SELNSVCPROG	*selection service*
100016	RDATABASEPROG	*remote database access*
100017	REXECPROG	*remote execution*
100018	ALICEPROG	*Alice Office Automation*
100019	SCHEDPROG	*scheduling service*

RPC Number	***Program***	***Description***
100020	LOCKPROG	*local lock manager*
100021	NETLOCKPROG	*network lock manager*
100022	X25PROG	*x.25 inr protocol*
100023	STATMON1PROG	*status monitor 1*
100024	STATMON2PROG	*status monitor 2*
100025	SELNLIBPROG	*selection library*
100026	BOOTPARAMPROG	*boot parameters service*
100027	MAZEPROG	*mazewars game*
100028	YPUPDATEPROG	*yp update*
100029	KEYSERVEPROG	*key server*
100030	SECURECMDPROG	*secure login*
100031	NETFWDIPROG	*nfs net forwarder init*
100032	NETFWDTPROG	*nfs net forwarder trans*
100033	SUNLINKMAP_PROG	*sunlink MAP*
100034	NETMONPROG	*network monitor*
100035	DBASEPROG	*lightweight database*
100036	PWDAUTHPROG	*password authorization*
100037	TFSPROG	*translucent file svc*
100038	NSEPROG	*nse server*
100039	NSE_ACTIVATE_PROG	*nse activate daemon*
150001	PCNFSDPROG	*pc passwd authorization*
200000	PYRAMIDLOCKINGPROG	*Pyramid-locking*
200001	PYRAMIDSYS5	*Pyramid-sys5*
200002	CADDS_IMAGE	*CV cadds_image*
300001	ADT_RFLOCKPROG	*ADT file locking*

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_wrapstring()
xdr_char()     xdr_u_char()
```

Note that the routine *xdr_string()* exists, but cannot be used with *callrpc()* and *registerrpc*(), which only pass two parameters to their XDR routines. *xdr_wrapstring()* has only two parameters, and is thus OK. It calls *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* section of this manual, only few implementation examples are given here.

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

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

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

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

and make an RPC call such as

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

with *xdr_varintarr()* defined as:

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

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, one can use *xdr_vector*(), which serializes fixed-length arrays.

```
int intarr[SIZE];

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

    return (xdr_vector(xdrsp, intarr, SIZE, sizeof(int),
        xdr_int));
}
```

XDR always converts quantities to 4-byte multiples when serializing. Thus, if either of the examples 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; *xdr_bytes()* has four parameters, similar to the first four parameters of *xdr_array*(). For null-terminated strings, there is also the *xdr_string()* routine, which is the same as *xdr_bytes()* without the length parameter. 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;
{

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

Note that we could as easily call *xdr_simple()* here instead of *xdr_reference*().

3. Lowest Layer 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.

There are several occasions when you may need to use lower layers of RPC. First, you may need to use TCP, since the higher layer uses UDP, which restricts RPC calls to 8K bytes of data. Using TCP permits calls to send long streams of data. For an example, see the *TCP* section below. Second, you may want to allocate and free memory while serializing or deserializing with XDR routines. There is no call at the higher level to let you free memory explicitly. For more explanation, see the *Memory Allocation with XDR* section below. Third, you may need to perform authentication on either the client or server side, by supplying credentials or verifying them. See the explanation in the *Authentication* section below.

3.1. More on the Server Side

The server for the *nusers()* program shown below does the same thing as the one using *registerrpc()* above, but is written using a lower layer of the RPC package:

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

main()
{
    SVCXPRT *transp;
    int nuser();

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

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

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

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

If the user specifies the *RPC_ANYSOCK* argument, the RPC library routines will open sockets. Otherwise they will expect the user to do so. The routines *svcudp_create()* and *clntudp_create()* will cause the RPC library routines to *bind()* their socket if it is not bound already.

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

After creating an *SVCXPRT*, the next step is to call *pmap_unset()* so that if the *nusers()* server crashed earlier, any previous trace of it is erased before restarting. More precisely, *pmap_unset()* erases the entry for *RUSERSPROG* from the 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 results. 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 receives 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, "can’t reply to RPC call\n");
         return (1);
    }
    return;
}
```

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

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));
}
```

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

```
char chararr[SIZE];

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

If you want XDR to do the allocation, you would have to rewrite 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(transp, xdr_chararr2, &arrptr);
```

Note that, after being used, the character array can be freed with *svc_freeargs()* *svc_freeargs()* will not attempt to free any memory if the variable indicating it is NULL. For example, in the the routine *xdr_finalexample*(), given earlier, if *finalp->string* was NULL, then it would not be freed. The same is true for *finalp->simplep*.

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from *callrpc()* the 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. See the *External Data Representation: Sun Technical Notes* for examples of more sophisticated XDR routines that determine which of the three modes they are in and adjust their behavior accordingly.

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 <utmp.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    unsigned long nusers;

    if (argc != 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(-1);
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
        fprintf(stderr, "can’t get addr for %s\n",argv[1]);
        exit(-1);
    }
    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
        hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clntudp_create(&server_addr, RUSERSPROG,
      RUSERSVERS, pertry_timeout, &sock)) == NULL) {
        clnt_pcreateerror("clntudp_create");
        exit(-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
        0, xdr_u_long, &nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(-1);
    }
    clnt_destroy(client);
    close(sock);
    exit(0);
}
```

The low-level version of *callrpc()* is *clnt_call()* which takes a *CLIENT* pointer rather than a host name. The parameters to *clnt_call()* are a *CLIENT* pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the return value, a pointer to where the return value will be placed, and the time in seconds to wait for a reply.

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

The parameters to *clntudp_create()* are the server address, the program number, the version number, a timeout value (between tries), and a pointer to a socket. 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.

Note that the *clnt_destroy()* call always deallocates the space associated with the *CLIENT* handle. It closes the socket associated with the *CLIENT* handle, however, only if the RPC library opened it. It the socket was opened by the user, it stays open. This makes it possible, in cases where there are multiple client handles using the same socket, to destroy one handle without closing the socket that other handles are using.

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

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

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

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

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

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()
{
    fd_set readfds;
    int dtbsz = getdtablesize();

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

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

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

4.2. Broadcast RPC

The *portmapper* is a daemon that converts RPC program numbers into DARPA protocol port numbers; see the *portmap* man page. You can't do broadcast RPC without the portmapper. 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.
5. Broadcast requests are limited in size to the MTU (Maximum Transfer Unit) of the local network. For Ethernet, the MTU is 1500 bytes.

4.2.1. Broadcast RPC Synopsis

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

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

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

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

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 nonbatched 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 <suntool/windows.h>

void windowdispatch();

main()
{
    SVCXPRT *transp;

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL){
        fprintf(stderr, "can’t create an RPC server\n");
        exit(1);
    }
    pmap_unset(WINDOWPROG, WINDOWVERS);
    if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
      windowdispatch, IPPROTO_TCP)) {
        fprintf(stderr, "can’t register WINDOW service\n");
        exit(1);
    }
    svc_run();  /* Never returns */
    fprintf(stderr, "should never reach this point\n");
}

void
windowdispatch(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    char *s = NULL;

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0))
            fprintf(stderr, "can’t reply to RPC call\n");
        return;
    case RENDERSTRING:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can’t decode arguments\n");
            /*
             * Tell caller he screwed up
             */
            svcerr_decode(transp);
            break;
        }
        /*
         * Code here to render the string s
         */
        if (!svc_sendreply(transp, xdr_void, NULL))
            fprintf(stderr, "can’t reply to RPC call\n");
        break;
    case RENDERSTRING_BATCHED:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can’t decode arguments\n");
            /*
             * We are silent in the face of protocol errors
             */
            break;
        }
        /*
         * Code here to render string s, but send no reply!
         */
        break;
    default:
        svcerr_noproc(transp);
        return;
    }
    /*
     * Now free string allocated while decoding arguments
     */
    svc_freeargs(transp, xdr_wrapstring, &s);
}
```

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 (EOF):

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

main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    char buf[1000], *s = buf;

    if ((client = clnttcp_create(&server_addr,
      WINDOWPROG, WINDOWVERS, &sock, 0, 0)) == NULL) {
        perror("clnttcp_create");
        exit(-1);
    }
    total_timeout.tv_sec = 0;
    total_timeout.tv_usec = 0;
    while (scanf("%s", s) != EOF) {
        clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
            xdr_wrapstring, &s, NULL, NULL, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
            clnt_perror(client, "batched rpc");
            exit(-1);
        }
    }

    /* Now flush the pipeline */

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

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

The above example was completed to render all of the (2000) lines in the file */etc/termcap*. The rendering service did nothing but 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.

4.4.1. UNIX Authentication

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 where client is */
    int     aup_uid;        /* client's UNIX effective uid */
    int     aup_gid;        /* client's current group id */
    u_int   aup_len;        /* element length of aup_gids */
    int     *aup_gids;      /* array of groups user is in */
};
```

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

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

This should be done in all cases, to conserve memory.

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 vers num */
    u_long    rq_proc;        /* desired procedure number */
    struct opaque_auth rq_cred; /* raw credentials from wire */
    caddr_t   rq_clntcred;   /* credentials (read only) */
};
```

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

```
/*
 * 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) fields 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, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authunix_parms *unix_cred;
    int uid;
    unsigned long nusers;

    /*
     * we don’t care about authentication for null proc
     */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can’t reply to RPC call\n");
            return (1);
         }
         return;
    }
    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_UNIX:
        unix_cred =
            (struct authunix_parms *)rqstp->rq_clntcred;
        uid = unix_cred->aup_uid;
        break;
    case AUTH_NULL:
    default:
        svcerr_weakauth(transp);
        return;
    }
    switch (rqstp->rq_proc) {
    case RUSERSPROC_NUM:
        /*
         * make sure caller is allowed to call this proc
         */
        if (uid == 16) {
            svcerr_systemerr(transp);
            return;
        }
        /*
         * Code here to compute the number of users
         * and assign it to the variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
            fprintf(stderr, "can’t reply to RPC call\n");
            return (1);
        }
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

A few things should be noted 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. DES Authentication

UNIX authentication is quite easy to defeat. Instead of using *authunix_create_default*(), one can call *authunix_create()* and then modify the RPC authentication handle it returns by filling in whatever user ID and hostname they wish the server to think they have. DES authentication is thus recommended for people who want more security than UNIX authentication offers.

The details of the DES authentication protocol are complicated and are not explained here. See *Remote Procedure Calls: Protocol Specification* for the details.

In order for DES authentication to work, the *keyserv(8c)* daemon must be running on both the server and client machines. The users on these machines need public keys assigned by the network administrator in the *publickey(5)* database. And, they need to have decrypted their secret keys using their login password. This automatically happens when one logs in using *login(1)*, or can be done manually using *keylogin(1)*. The *Network Services* chapter explains more how to setup secure networking.

Client Side

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

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

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

```
char servername[MAXNETNAMELEN];

host2netname(servername, rhostname, NULL);
```

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

```
char servername[MAXNETNAMELEN];

user2netname(servername, getuid(), NULL);
```

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

The second argument to *authdes_create()* is a lifetime for the credential. Here it is set to sixty seconds. What that means is that the credential will expire 60 seconds from now. If some mischievous user tries to reuse the credential, the server RPC subsystem will recognize that it has expired and not grant any requests. If the same mischievous user tries to reuse the credential within the sixty second lifetime, he will still be rejected because the server RPC subsystem remembers which credentials it has already seen in the near past, and will not grant requests to duplicates.

The third argument to *authdes_create()* is the address of the host to synchronize with. In order for DES authentication to work, the server and client must agree upon the time. Here we pass the address of the server itself, so the client and server will both be using the same time: the server's time. The argument can be *NULL*, which means "don't bother synchronizing." You should only do this if you are sure the client and server are already synchronized.

The final argument to *authdes_create()* is the address of a DES encryption key to use for encrypting timestamps and data. If this argument is *NULL*, as it is in this example, a random key will be chosen. The client may find out the encryption key being used by consulting the *ah_key* field of the authentication handle.

Server Side

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

```
#include <sys/time.h>
#include <rpc/auth_des.h>
    . . .
    . . .
nuser(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authdes_cred *des_cred;
    int uid;
    int gid;
    int gidlen;
    int gidlist[10];
    /*
     * we don't care about authentication for null proc
     */

    if (rqstp->rq_proc == NULLPROC) {
        /* same as before */
    }

    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_DES:
        des_cred =
            (struct authdes_cred *) rqstp->rq_clntcred;
        if (! netname2user(des_cred->adc_fullname.name,
            &uid, &gid, &gidlen, gidlist))
        {
            fprintf(stderr, "unknown user: %s0,
                des_cred->adc_fullname.name);
            svcerr_systemerr(transp);
            return;
        }
        break;
    case AUTH_NULL:
    default:
        svcerr_weakauth(transp);
        return;
    }

    /*
     * The rest is the same as before
     */
```

Note the use of the routine *netname2user*(), the inverse of *user2netname*(): it takes a network ID and converts to a unix ID. *netname2user*() also supplies the group IDs which we don't use in this example, but which may be useful to other UNIX programs.

4.6. Using Inetd

An RPC server can be started from *inetd* The only difference from the usual code is that the service creation routine should be called in the following form:

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

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

```
svc_register(transp, PROGNUM, VERSNUM, service, 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/inetd.conf* for RPC services is in one of the following two forms:

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

where *p_name* is the symbolic name of the program as it appears in *rpc(5)*, *server* is the program implementing the server, and *program* and *version* are the program and version numbers of the service. For more information, see *inetd.conf(5)*.

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

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

5. More Examples

5.1. Versions

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

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

Both versions can be handled by the same C procedure:

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

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

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[BUFSIZ], *p;

    if (xdrs->x_op == XDR_FREE)/* nothing to free */
        return 1;
    while (1) {
        if (xdrs->x_op == XDR_ENCODE) {
            if ((size = fread(buf, sizeof(char), BUFSIZ,
              fp)) == 0 && ferror(fp)) {
                fprintf(stderr, "can’t fread\n");
                return (1);
            }
        }
        p = buf;
        if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
            return 0;
        if (size == 0)
            return 1;
        if (xdrs->x_op == XDR_DECODE) {
            if (fwrite(buf, sizeof(char), size,
              fp) != size) {
                fprintf(stderr, "can’t fwrite\n");
                return (1);
            }
        }
    }
}
```

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

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

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

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

    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "can’t get addr for ’%s’\n", host);
        return (-1);
    }
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
        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");
        return (-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum,
        inproc, in, outproc, out, total_timeout);
    clnt_destroy(client);
    return (int)clnt_stat;
}

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

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

    if ((transp = svctcp_create(RPC_ANYSOCK,
      BUFSIZ, BUFSIZ)) == NULL) {
        fprintf("svctcp_create: error\n");
        exit(1);
    }
    pmap_unset(RCPPROG, RCPVERS);
    if (!svc_register(transp,
      RCPPROG, RCPVERS, rcp_service, IPPROTO_TCP)) {
        fprintf(stderr, "svc_register: error\n");
        exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "svc_run should never return\n");
}

rcp_service(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (svc_sendreply(transp, xdr_void, 0) == 0) {
            fprintf(stderr, "err: rcp_service");
            return (1);
        }
        return;
    case RCPPROC_FP:
        if (!svc_getargs(transp, xdr_rcp, stdout)) {
            svcerr_decode(transp);
            return;
        }
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can’t reply\n");
            return;
        }
        return (0);
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

5.3. Callback Procedures

Occasionally, it is useful to have a server become a client, and make an RPC call back to 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 proto, vers, *sockp;
{
    static int prognum = 0x40000000;
    int s, len, socktype;
    struct sockaddr_in addr;

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

Note: *The call to ntohs() is necessary to ensure that the port number in addr.sin_port, which is in network byte order, is passed in host byte order (as pmap_set() expects). See the byteorder(3N) man page for more details on the conversion of network addresses from network to host byte order.*

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 *EXAMPLEPROG* so that it can receive the RPC call informing it of the callback program number. Then at some random time (on receiving an *ALRM* signal in this example), it sends a callback RPC call, using the program number it received earlier.

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

int callback();
char hostname[256];

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

    gethostname(hostname, sizeof(hostname));
    s = RPC_ANYSOCK;
    x = gettransient(IPPROTO_UDP, 1, &s);
    fprintf(stderr, "client gets prognum %d\n", x);
    if ((xprt = svcudp_create(s)) == NULL) {
      fprintf(stderr, "rpc_server: svcudp_create\n");
        exit(1);
    }
    /* protocol is 0 - gettransient does registering
     */
    (void)svc_register(xprt, x, 1, callback, 0);
    ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
        EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);
    if ((enum clnt_stat) ans != RPC_SUCCESS) {
        fprintf(stderr, "call: ");
        clnt_perrno(ans);
        fprintf(stderr, "\n");
    }
    svc_run();
    fprintf(stderr, "Error: svc_run shouldn’t return\n");
}

callback(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
        case 0:
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: exampleprog\n");
                return (1);
            }
            return (0);
        case 1:
            if (!svc_getargs(transp, xdr_void, 0)) {
                svcerr_decode(transp);
                return (1);
            }
            fprintf(stderr, "client got callback\n");
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: exampleprog");
                return (1);
            }
    }
}

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

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

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

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

docallback()
{
    int ans;

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