Remote Procedure Call (RPC) Reference Manual

SR–2089 9.0

Cray Research, Inc.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

Requests for copies of Cray Research, Inc. publications should be sent to the following address:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120
USA

Order desk +1–612–683–5907
Fax number +1–612–452–0141

New Features

Remote Procedure Call (RPC) Reference Manual SR–2089 9.0

This version of the *Remote Procedure Call (RPC) Reference Manual* supports the UNICOS 9.0 release. The following changes have been made:

- Kerberos authentication flavor is documented in Section 3 and in Appendix E.
- The UNICOS multilevel security system notes have been modified.

Record of Revision

The date of printing or software version number is indicated in the footer. Changes in rewrites are noted by revision bars along the margin of the page.

Version	***Description***
	January 1989 – Original printing. This manual contains remote procedure call (RPC) information to support CRAY Y-MP, CRAY X-MP EA, CRAY X-MP, and CRAY-2 systems running the Cray operating system UNICOS release 5.0.
6.0	November 1990. This documentation supports the UNICOS 6.0 release. In addition to minor technical and organizational changes, documentation for the Data Encryption Standard (DES) type of RPC authentication has been added.
8.0	December 1993. This documentation supports the UNICOS 8.0 release running on all Cray Research systems. In addition to minor technical and organizational changes, documentation for the RPC service library routines was added.
8.1	April 1994. Updated only online to include editorial changes.
8.2	October 1994. Updated only online. This documentation supports the UNICOS 8.2 release running on all Cray Research systems. Documentation for Kerberos authentication was added.
8.3	November 1994. Updated only online. This documentation supports the UNICOS 8.3 release running on all Cray Research systems. One reference to a manual was added.
9.0	April 1995. This documentation supports the UNICOS 9.0 release running on all Cray Research systems. This version marks with change bars all technical changes made since the last printing at the UNICOS 8.0 release.

In this version, change bars indicate all technical changes made since the document was last printed for the UNICOS 8.0 release.

Preface

This publication documents the Cray Research, Inc. (CRI) implementation of Sun Microsystems' remote procedure call (RPC) facility for all Cray Research systems. Using the External Data Representation (XDR) data definition language, RPC provides the means for communicating across diverse network environments. These procedures are a standard part of the UNICOS operating system. This RPC facility can interface with any network file system (NFS) implementation.

Readers of this manual should be familiar with the C programming language and with the administration of User Datagram Protocol/Internet Protocol (UDP/IP) and Transmission Control Protocol/Internet Protocol (TCP/IP) networks in a Berkeley UNIX environment.

> **Note:** The Trusted UNICOS system is a configuration of the UNICOS MLS system that supports processing at multiple security labels and system administration using only non-super user administrative roles. The Trusted UNICOS system consists of the subset of UNICOS software that offers these capabilities. The Trusted UNICOS name does not imply maintenance of the UNICOS 8.0.2 security evaluation.
>
> For the UNICOS 10.0 release, the functionality of the Trusted UNICOS system will be retained, but the `CONFIG_TRUSTED` option, which enforces conformance to the strict B1 configuration, will no longer be available. All references to the Trusted UNICOS system will be removed from the UNICOS 10.0 documentation. See the *UNICOS 9.0 Release Overview*, RO–5000 9.0, for more information.

Related publications

The following documents contain additional information that may be helpful:

- *UNICOS User Commands Reference Manual*, publication SR–2011
- *UNICOS Networking Facilities Administrator's Guide*, publication SG–2304
- *UNICOS Administrator Commands Reference Manual*, publication SR–2022
- *ONC+ Technology for the UNICOS Operating System*, publication SG–2169

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software manuals that are available to customers.

To order a manual, either call the Distribution Center in Mendota Heights, Minnesota, at +1–612–683–5907 or send a facsimile of your request to fax number +1–612–452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users).

Conventions

The following conventions are used throughout this manual:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers: 1 User commands 1B User commands ported from BSD 2 System calls 3 Library routines, macros, and opdefs 4 Devices (special files) 4P Protocols 5 File formats 7 Miscellaneous topics 7D DWB-related information 8 Administrator commands
`routine`()	Routine names followed by an empty set of parentheses designate a library or kernel routine; for example, `ddcntl`(). Kernel routines do not have man pages associated with them.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command line.
...	Ellipses indicate that a preceding command-line element can be repeated.

Convention	Meaning
KEY	This convention indicates a key on the keyboard.
<KEY>	On man pages, this convention indicates a key on the keyboard.

The following machine naming conventions may be used throughout this manual:

Term	Definition
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems, including the following: CRAY C90 series (CRAY C916, CRAY C92A, CRAY C94, CRAY C94A, and CRAY C98 systems) CRAY C90D series (CRAY C92AD, CRAY C94D, and CRAY C98D systems) CRAY EL series (CRAY Y-MP EL, CRAY EL92, CRAY EL94, and CRAY EL98 systems) CRAY J90 series (CRAY J916 and CRAY J932 systems) CRAY T90 series (CRAY T94, CRAY T916, and CRAY T932 systems) CRAY Y-MP E series (CRAY Y-MP 2E, CRAY Y-MP 4E, CRAY Y-MP 8E, and CRAY Y-MP 8I systems) CRAY Y-MP M90 series (CRAY Y-MP M92, CRAY Y-MP M94, and CRAY Y-MP M98 systems)
Cray MPP systems	All configurations of Cray massively parallel processing (MPP) systems, including the CRAY T3D series (CRAY T3D MC, CRAY T3D MCA, and CRAY T3D SC systems)

Term	Definition
All Cray Research systems	All configurations of Cray PVP and Cray MPP systems that support this release
SPARC systems	All SPARC platforms that run the Solaris operating system version 2.3 or later

The default shell in the UNICOS 9.0 release, referred to in Cray Research documentation as the standard shell, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Company Standard XPG4

The UNICOS 9.0 operating system also supports the optional use of the C shell.

The POSIX standard uses *utilities* to refer to executable programs that Cray Research documentation usually refers to as *commands*. Both terms appear in this document.

In this publication, *Cray Research*, *Cray*, and *CRI* refer to Cray Research, Inc. and/or its products.

Online information

The following types of online information products are available to Cray Research customers:

- CrayDoc online documentation reader, which lets you see the text and graphics of a manual online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Docview text-viewer system, which lets you see the text of a manual online. The Docview system is available on the Cray Research mainframe. To start the Docview system, use the `docview`(1) command.
- Man pages, which describe a particular element of the UNICOS operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.

- UNICOS message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.
- Cray Research online glossary, which explains the terms used in a manual. To get a definition, use the `define`(1) command.
- `xhelp` help facility. This online help system is available within tools such as the Program Browser (`xbrowse`) and the MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

If you have comments about the technical accuracy, content, or organization of this manual, please tell us. You can contact us in any of the following ways:

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!publications`

- Send us electronic mail from any system connected to Internet, using the following Internet addresses:

  `pubs2089@timbuk.cray.com` (comments on this manual)

  `publications@timbuk.cray.com` (general comments)

- Contact your Cray Research representative and ask that a Software Problem Report (SPR) be filed. Use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.
- Call our Software Publications Group in Eagan, Minnesota, through the Technical Support Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600

- Send a facsimile of your comments to the attention of "Software Publications Group" in Eagan, Minnesota, at fax number +1–612–683–5599.
- Use the postage-paid Reader's Comment Form at the back of the printed manual.

We value your comments and will respond to them promptly.

Contents

	Page
Preface	iii
Related publications	iv
Conventions	v
Online information	vii
Reader comments	viii
Overview [1]	1
RPC and XDR	2
Identifying remote procedures	3
Remote program number	3
Remote program version number	6
Remote procedure number	8
Registering with the portmapper	8
Transports and semantics	8
Error messages	9
Remote Procedure Call (RPC) Programming [2]	11
Registering the routine on the server	12
Client call and server reply process	14
RPC layers	16
Highest RPC layer	17
Intermediate RPC layer	18
Registering in the intermediate layer	19
Calling and replying in the intermediate layer	20
Using XDR routines	21
XDR memory allocation	24
Lowest RPC layer	26
Registering in the lowest layer	26
Calling in the lowest layer	30
Select processing	33
TCP processing	34
Callback processing	39
Other uses of the RPC protocol	44
Batching	45
Broadcast RPC	50
Authentication [3]	53
Setting up authentication	54
Null authentication requirements	54
UNICOS authentication requirements	55
DES authentication requirements	55
Kerberos authentication requirements	56
Client authentication	56
Server authentication	57
Record-marking standard	61
RPC Message Protocol [4]	63
Call and reply	63
Message structure	64
Synopsis of RPC and XDR Routines [A]	67
rpcgen Programming Guide [B]	97
External Data Representation Standard: Protocol Specification [C]	121

Page

Remote Procedure Calls: Protocol Specification [D] 141

Authentication Routines [E] 165

UNICOS authentication 165

DES authentication 166

Kerberos authentication 171

Service Library Routines [F] 177

intro_svc(3R) 179

getrpcport(3R) 180

havedisk(3R) 181

publickey(3R) 182

rnusers(3R) 183

rstat(3R) 185

rwall(3R) 186

yppasswd(3R) 187

Index 189

Figure

Figure 1. RPC paradigm 2

Tables

Table 1. Remote program number assignments 3

Table 2. Registered program list 4

Table 3. RPC service library routines 18

Overview [1]

Note: The RPC feature is included in the trusted computing base (TCB) of the Trusted UNICOS system. The operator functions, procedures, and duties outlined in the following subsections are required to maintain a Trusted UNICOS system. No special security administrator or operator functions are necessary for the management of RPC on a Trusted UNICOS system.

Remote procedure calls (RPCs) provide a way for you do the following:

- Distribute program segments across computers in a network
- Communicate with more than one machine on a given network while executing a program
- Communicate with other programs that run on the same machine

The typical configuration for environments that use RPCs consists of workstations connected to a computer through a network. The workstations are used for application interface and high-resolution displays; the computationally intense part of the code runs on the computer.

A program must be registered so that other programs on the network can find it. (See an example of registering in subsection 2.1, page 12.) RPCs use a client/server paradigm in which the client first sends data to a server running in a machine on a network. The server receives the data packet, processes it as required, and returns a result to the client. The server does not have to return any information to the client. In C language context, the server can be a function of type `void`. The same is true for the client; it can call a server without passing data to it. Figure 1 demonstrates the typical RPC paradigm.

RPC and XDR
1.1

The machines on a given network can run in different operating system environments. Programs that use RPC are shielded from the calling conventions of these various operating systems by the use of data translation routines known as External Data Representation (XDR) routines. XDR is a protocol that allows programmers to specify arbitrary data structures that are independent of a specific machine. These routines ensure that data of any type can be passed successfully between machines with potentially different word sizes or other architectural differences.

XDR routines act as filters for the data moving back and forth, ensuring that the data is translated into a form that the receiving machine can interpret correctly. Translating data from the sending machine into XDR format is called *serializing*. When the receiving machine interprets the serialized data, the process is called *deserializing*. The XDR routines do the serializing and deserializing for each communication between server and client.

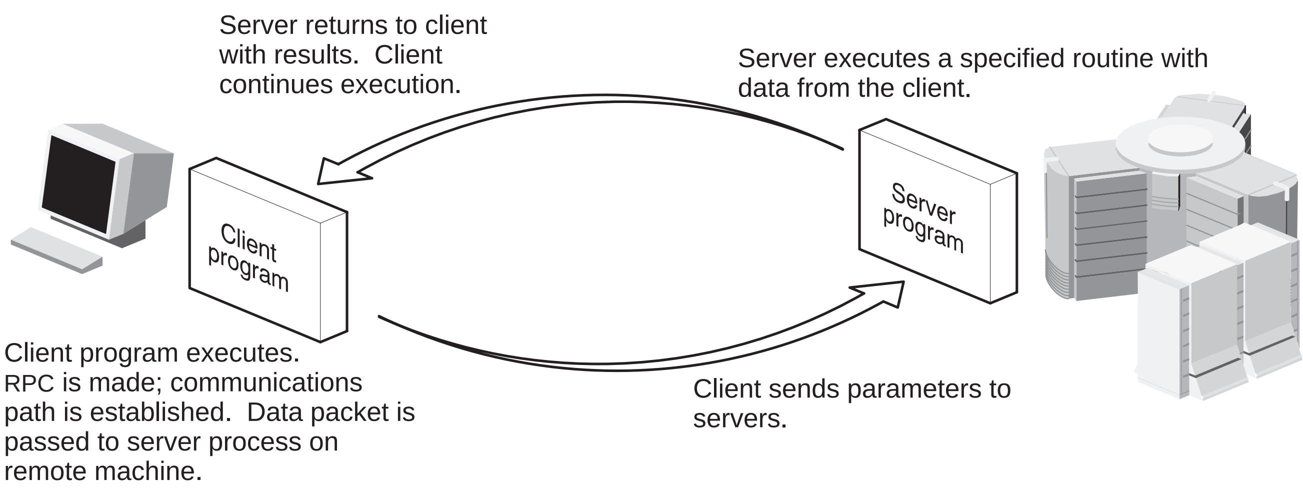

Figure 1. RPC paradigm

The C library (`libc.a`) contains predefined XDR routines for passing most data types. If only one request value is being passed to the server and one result value is being returned, XDR routines from the library can be used. However, when a structure is being passed, the programmer must construct an XDR routine that maps the structure members to predefined XDR routines by member type. The `rpcgen` utility can automate the writing of structure XDR conversion routines. See appendix B, page 97, for information on `rpcgen`. Of course, if the data going to and from the client and server is the same in type (for example, the client passes a structure of two integers to the server, and the server passes a structure of two integers back to the client), they can share the same user-developed XDR routine.

RPC and XDR, based on RFC 1057 and RFC 1014, respectively, have been placed in the public domain; they serve as a standard for network application development.

Identifying remote procedures
1.2

An RPC message has three unsigned fields: the remote program number, the remote program version number, and the remote procedure number. These fields uniquely identify the procedure to be called.

Remote program number
1.2.1

The user's remote program number is a unique number in the range 0x20000000 to 0x3fffffff. Numbers outside of this range are reserved for other uses (see Table 1 for assigned ranges).

Table 1. Remote program number assignments

Program number	Assignment
0x0 – 0x1fffffff	Sun
0x20000000 – 0x3fffffff	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 RPC users. If a user develops an application that is of general interest, that application can be given an assigned number in the first range. The second group of numbers is reserved for specific customer applications and will not, in general, be the same across machines. This range is intended primarily for debugging new programs. The third group is reserved for programs that generate program numbers dynamically. The final groups are reserved for future use and should not be used by any user-developed programs.

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

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

Please include a compilable `rpcgen`*.x* file that describes your protocol. In return, you will be given a unique program number.

You can find the RPC program numbers and protocol specifications of standard RPC services in the include files in the `/usr/include/rpcsvc` directory. These services, however, constitute only a small subset of those that have already been registered. Table 2 contains the most recent list of registered programs, as of the time of this printing. An asterisk denotes programs that are provided in the UNICOS 9.0 software release.

Table 2. Registered program list

RPC number	Program	Description
100000*	`PMAPPROG`	Portmapper
100001*	`RSTATPROG`	Remote statistics
100002*	`RUSERSPROG`	Remote users
100003*	`NFSPROG`	NFS daemon
100004*	`YPPROG`	Network information service (NIS)
100005*	`MOUNTPROG`	`mount` daemon
100006	`DBXPROG`	Remote `dbx`
100007*	`YPBINDPROG`	NIS binder
100008*	`WALLPROG`	Shutdown message
100009*	`YPPASSWDPROG`	NIS password server
100010	`ETHERSTATPROG`	Ethernet statistics server

Table 2. Registered program list (continued)

RPC number	Program	Description
100011	RQUOTAPROG	Disk quota server
100012*	SPRAYPROG	Spray packets server
100013	IBM3270PROG	3270 mapper
100014	IBMRJEPROG	RJE mapper
100015	SELNSVCPROG	Selection service
100016	RDATABASEPROG	Remote database access
100017	REXECPROG	Remote execution server
100018	ALICEPROG	Alice office automation
100019	SCHEDPROG	Scheduling service
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	NIS update server
100029*	KEYSERVEPROG	Key server for secure RPC
100030	SECURECMDPROG	Secure login
100031	NETFWDIPROG	NFS net forwarder initializing
100032	NETFWDTPROG	NFS net forwarder transmission
100033	SUNLINKMAP_PROG	SunLink MAP
100034	NETMONPROG	Network monitor
100035	DBASEPROG	Lightweight database
100036	PWDAUTHPROG	Password authorization
100037	TFSPROG	Translucent file service
100038	NSEPROG	NSE server
100039	NSE_ACTIVATE_PRG	NSE activate daemon

Table 2. Registered program list (continued)

RPC number	Program	Description
150001*	`PCNFSDPROG`	PC password authentication
200000	`PYRAMIDLOCKINGPROG`	Pyramid-locking
200001	`PYRAMIDSYS5`	Pyramid-sys5
200002	`CADDS_IMAGE`	CV `cadds_image`
300001	`ADT_RFLOCKPROG`	ADT file locking

Remote program version number 1.2.2

The version number is the release number for the RPC procedure. By convention, the first version number of any program *PROG* is *PROG*`VERS_ORIG`, and the most recent version is *PROG*`VERS`. In the following example, *PROG*=`RUSERS`. Suppose a new version of the user program returns an unsigned short rather than a long value. If this version is named `RUSERSVERS_SHORT`, a server that wants to support the first version and this version would register twice, as follows:

```
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);
}
```

The same C procedure can handle both versions, as follows:

```
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, “can’t reply to RPC call\n”);
                }
                return;

        case RUSERSPROC_NUM:
                /*
                 * Code here to compute the number of users
                 * and put it in the variable ’nusers’
                 */

                if (rqstp->rq_vers == RUSERSVERS_ORIG) {
                        if (!svc_sendreply(transp, xdr_u_long, &nusers) {
                               fprintf(stderr,“can’t reply to RPC call\n”);
                        }
                } else if (rqstp->rq_vers == RUSERSVERS_SHORT) {
                        nusers2 = (unsigned short) nusers;
                        if (!svc_sendreply(transp, xdr_u_short, &nusers2) {
                               fprintf(stderr,“can’t reply to RPC call\n”);
                       }
                } else {
                  /* send “bad version” error reply */
            svcerr_progvers(transp, RUSERSVERS_ORIG, RUSERSVERS_SHORT);
                }
                return;

                default:
            /* send “bad procedure” error reply */
                svcerr_noproc(transp);
                return;
        } /* end switch */
}
```

Remote procedure number
1.2.3

The procedure number is the number of the routine being referenced in the server; it identifies the procedure to be called. Servers can register many RPC routines, which would typically be numbered in order, 1, 2, 3, ... *n*. Procedure numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification might state that its procedure number 5 is read, and its procedure number 12 is write.

Registering with the portmapper
1.2.4

RPC servers can register themselves with the portmapper. This capability is useful if, for some reason, it becomes necessary to restart the portmapper while the RPC servers continue. The `–r` option of the `portmap` command specifies restart for standard RPC servers. The `–f` option directs the portmapper to send a `SIGHUP` signal to each of the process/UID pairs found in the file. See `portmap`(8) for a list of the servers that can be restarted and for examples of the `-r` and `–f` parameters.

Transports and semantics
1.3

The RPC protocol deals only with the specification and interpretation of messages; it is independent of transport protocols. Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Some semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol.

For example, when UDP/IP is used, RPC message passing is unreliable. Thus, if the client retransmits call messages after short time-outs, it can only infer from no reply message that the remote procedure was executed zero or more times (and from a reply message, one or more times). On the other hand, when TCP/IP is used, RPC message passing is reliable. No reply message means that the remote procedure was executed at most once; a reply message means that the remote procedure was executed exactly once.

Error messages
1.4

The reply message to a request message contains enough information to distinguish the following error conditions:

- The remote implementation of RPC is not compatible with protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is unavailable on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist (this is usually a client-side protocol or programming error).
- The parameters to the remote procedure are invalid from the server’s perspective. (Again, this is usually caused by a difference in the protocol between client and server.)
- An authentication error occurred.

Remote Procedure Call (RPC) Programming [2]

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

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

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

Example 1:

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

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

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

Registering the routine on the server
2.1

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

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

Example 1A:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Client call and server reply process
2.2

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

Example 1B:

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

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

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

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

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

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

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

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

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

```
RPC: Program not registered
```

RPC layers
2.3

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

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

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

Highest RPC layer 2.3.1

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

Example 2:

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

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

```
% cc program.c –lrpcsvc
```

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

Table 3. RPC service library routines

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

Intermediate RPC layer
2.3.2

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

Registering in the intermediate layer 2.3.2.1

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

Example 3:

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

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

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

Calling and replying in the intermediate layer 2.3.2.2

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

Example 4:

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

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

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

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

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

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

Using XDR routines 2.3.2.3

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

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

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

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

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

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

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

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

You could then make the following RPC:

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

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

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

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

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

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

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

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

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

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

and call `callrpc`, as follows:

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

Write `xdr_simple()`, as follows:

```
#include <rpc/rpc.h>
xdr_simple(xdrsp, simplep)
    XDR *xdrsp;
    struct simple *simplep;
{
    if (!xdr_int(xdrsp, &simplep->a))
        return (0);
    if (!xdr_short(xdrsp, &simplep->b))
        return (0);
    return (1);
}
```

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

Example 5:

```
typedef struct finalexample {
    char *string;
    struct simple *simplep;
} finalexample;

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

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

XDR memory allocation 2.3.2.4

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

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

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

It might be called from a server, as follows:

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

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

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

Then the RPC might look like this:

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

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

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

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

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

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

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

Lowest RPC layer 2.3.3

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

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

Registering in the lowest layer 2.3.3.1

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

Example 6:

```
 #include <stdio.h>
 #include <rpc/rpc.h>
 #include <rpcsvc/rusers.h>
 main()
 {
     SVCXPRT *transp;
     int nuser();
     transp=svcudp_create(RPC_ANYSOCK);
     if (transp == NULL){
        fprintf(stderr, “can’t create an RPC server\n”);
        exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS,
             nuser, IPPROTO_UDP)) {
        fprintf(stderr, “can’t register RUSER service\n”);
        exit(1);
    }
    svc_run(); /* never returns */
    fprintf(stderr, “should never reach this point\n”);
}
nuser(rqstp, tranp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, “can’t reply to RPC call\n”);
           return;
        }
        return;
    case RUSERSPROC_NUM:
        /*
         * code here to compute the number of users
         * and put in variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers) {
            fprintf(stderr, “can’t reply to RPC call\n”);
           return;
        }
        return;
    default:
        svcerr_noproc(transp);
        return;
    } }
```

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

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

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

1. When a server starts up, it advertises to a portmapper daemon on its local machine.

2. The server-side portmap daemon picks a port number for the RPC procedure if the socket specified as a parameter to `svcudp_create` is not already bound.

3. On the client side, when the `clntudp_create` call is made with an unbound socket, the system queries the portmapper on the machine to which the call is being made, and it gets the appropriate port number.

4. If the portmapper is not running on the server side, or has no port that corresponds to the RPC, the RPC fails.

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

Lines 13 through 17: After creating a service transport handle, `(SVCXPRT)`, the next step is to call `pmap_unset` so that, if the `nusers` server crashed earlier, any previous trace of it is erased before restarting. More precisely, `pmap_unset` erases the entry for `RUSERSPROG` from the portmapper’s tables.

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

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

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

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

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

Example 7:

```
case RUSERSPROC_BOOL: {
    int bool;
    unsigned nuserquery;
    if (!svc_getargs(transp, xdr_u_int, &nuserquery) {
        svcerr_decode(transp);
        return;
    }
    /*
     * code to set nusers = number of users
     */
    if (nuserquery == nusers)
        bool = TRUE;
    else
        bool = FALSE;
    if (!svc_sendreply(transp, xdr_bool, &bool){
         fprintf(stderr, “can’t reply to RPC call\n”);
         exit(1);
    }
    return;
}
```

Calling in the lowest layer 2.3.3.2

When you use `callrpc`, you have no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate the layer of RPC that lets you adjust these parameters, consider example 8, which contains code to call the `nusers` service.

Example 8:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int addrlen, sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    unsigned long nusers;
    if (argc < 2) {
        fprintf(stderr, “usage: nusers hostname\n”);
        exit(–1);
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
        fprintf(stderr, “can’t get addr for %s\n”,argv[1]);
        exit(–1);
    }
    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
    addrlen = sizeof(struct sockaddr_in);
    bzero ((char*) &server_addr, sizeof (server_addr));
    bcopy(hp–>h_addr, (caddr_t)&server_addr.sin_addr,
        hp–>h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = 0;
   if ((client = clntudp_create(&server_addr, RUSERSPROG,
     RUSERSVERS, pertry_timeout, &sock)) == NULL) {
        clnt_pcreateerror(“clntudp_create”);
        exit(–1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void,
        0, xdr_u_long, &nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, “rpc”);
        exit(–1);
    }
    clnt_destroy(client);
    close(sock);
    exit(0)
    }
```

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

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

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

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

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

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

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

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

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

```
struct CLIENT *cp;

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

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

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

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

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

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

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

Select processing 2.3.3.3

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

Example 9:

```
void
svc_run()
{
    fd_set readfds;
    extern int errno;
    for (;;) {
    readfds = svc_fdset;
    switch (select(32, &readfds, NULL, NULL, NULL)) {
        case –1:
        if (errno == EINTR)
                continue;
            perror(“select”);
            return;
        case 0:
            break;
        default:
            svc_getreqset(&readfds);
        }
    }
}
```

You can bypass `svc_run` and call `svc_getreq` (or `svc_getreqset`) yourself. To do so, you must know only the file descriptors of the sockets associated with the programs for which you are waiting. Thus, you can have your own `select`(2), which waits on both the RPC socket and your own descriptors.

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

TCP processing
2.3.3.4

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

Example 10:

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

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

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

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

        if (xdrs–>x_op == XDR_FREE) {
                return(1);
        }

        while (1) {
                if (xdrs–>x_op == XDR_ENCODE) {
                if ((size = fread(buf, sizeof(char), BUFSIZ, fp) == 0)
                        && ferror(fp)) {
                        fprintf(stderr, “can’t fread”\n”);
                        exit(1);
                        }
```

(continued)

```
                }

                p = buf;

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

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


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

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

/*
* The sender routines
 */

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

int callrpctcp();

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

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

(continued)

```
        }

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

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

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

        bzero((char*)&server_addr, sizeof(server_addr));
        bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = 0;

        if ((client = clnttcp_create(&server_addr, prognum, versnum,
                &sock, BUFSIZ, BUFSIZ)) == NULL ) {

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

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

(continued)

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

/*
 * The receiving routines
 */

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

main()
{
       register SVCXPRT *transp;

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

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

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

                fprintf(stderr, "svc_register: error\n");
                exit(1);
       }
       svc_run();            /* should never return */
       fprintf(stderr, "svc_run should not return, but it did!\n");
}

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

       case NULLPROC:
                if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, "err: rcp NULL service\n");
                }
                return;
```

(continued)

```
        case RCPPROC_FP:
                 if (!svc_getargs(transp, xdr_rcp, stdout)) {
                        svcerr_decode(transp);
                        return;
                 }
                 if (!svc_sendreply(transp, xdr_void, 0)) {
                        fprintf(stderr, “can’t send reply\n”);
                 }
                 return;

        default:
                 svcerr_noproc(transp);
                 return;

                 }       /* end switch */
}
```

Callback processing
2.4

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

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

Example 11:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
gettransient(proto, vers, sockp)
   int proto, vers, *sockp;
{
   static int prognum = 0x40000000;
   int s, len, socktype;
   struct sockaddr_in addr;
   switch(proto) {
       case IPPROTO_UDP:
           socktype = SOCK_DGRAM;
           break;
       case IPPROTO_TCP:
           socktype = SOCK_STREAM;
           break;
       default:
           fprintf(stderr, “unknown protocol type\n”);
           return 0;
   }
   if (*sockp == RPC_ANYSOCK) {
       if ((s = socket(AF_INET, socktype, 0)) <0) {
           perror(“socket”);
           return (0);
       }
       *sockp = s;
   }
   else
       s = *sockp;
   bzero ((char*) &addr, sizeof (addr));
   addr.sin_addr.s_addr = 0;
   addr.sin_family = AF_INET;
   addr.sin_port = 0;
   len = sizeof(addr);
   /*
    * may be already bound, so don’t check for error
    */
   bind(s, &addr, len);
   if (getsockname(s, &addr, &len)< 0) {
       perror(“getsockname”);
       return (0);
   }
   while (!pmap_set(prognum++, vers, proto, addr.sin_port))
       continue;
   return (prognum–1);
)
```

The two programs in example 12 illustrate how to use the `gettransient` routine. The client makes an RPC to the server, passing it a transient program number. The client then waits to receive a callback from the server at that program number. The server registers the program EXAMPLEPROG, so that it can receive the RPC informing it of the callback program number. Then at some random time (on receiving an ALRM signal in this example), it sends a callback RPC, using the program number it received earlier.

Example 12:

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>
int callback();
char hostname[256];
main(argc, argv)
   int argc;
   char **argv;
{
   int x, ans, s;
   SVCXPRT *xprt;
   gethostname(hostname, sizeof(hostname));
   s = RPC_ANYSOCK;
   x = gettransient(IPPROTO_UDP, 1, &s);
   fprintf(stderr, “client gets prognum %d\n”, x);
   if ((xprt = svcudp_create(s)) == NULL) {
    fprintf(stderr, “rpc_server: svcudp_create\n”);
       exit(1);
   }
   /* protocol is 0 – gettransient() does registering
    */
   (void)svc_register(xprt, x, 1, callback, 0);
   ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
       EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);
   if (ans != RPC_SUCCESS) {
       fprintf(stderr, “call: ”);
       clnt_perrno(ans);
       fprintf(stderr, ”\n”);
       exit(1)
```

(continued)

```
   }
   svc_run();
   fprintf(stderr, “Error: svc_run shouldn’t return\n”);
}
callback(rqstp, transp)
   register struct svc_req *rqstp;
   register SVCXPRT *transp;
{
   switch (rqstp–>rq_proc) }
       case 0:
           if (!svc_sendreply(transp, xdr_void, 0)) {
               fprintf(stderr, “err: rusersd\n”);
               exit(1);
           }
           exit(0);
       case 1:
           if (!svc_getargs(transp, xdr_void, 0)) {
               svcerr_decode(transp);
               exit(1);
           }
           fprintf(stderr, “client got callback\n”);
           if (!svc_sendreply(transp, xdr_void, 0)) {
               fprintf(stderr, “err: rusersd”);
               exit(1);
           }
   }
}
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>
char *getnewprog();
char hostname[256];
int docallback();
int pnum;        /* program number for callback routine */
main(argc, argv)
int argc
   char **argv;
{
```

(continued)

```
   gethostname(hostname, sizeof(hostname));
   registerrpc(EXAMPLEPROG, EXAMPLEVERS,
    EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
   fprintf(stderr, “server going into svc_run\n”);
   signal(SIGALRM, docallback);
   alarm(10);
   svc_run();
   fprintf(stderr, “Error: svc_run shouldn’t return\n”);
}
char *
getnewprog(pnump)
   char *pnump;
{
   pnum = *(int *)pnump;
   return NULL;
}
docallback()
{
   int ans;
   ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
       xdr_void, 0);
   if (ans != 0) {
       fprintf(stderr, “server:\n”);
       clnt_perrno(ans);
       fprintf(stderr, ”\n”);
   }
}
```

Other uses of the RPC protocol
2.5

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

Batching
2.5.1

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

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

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

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

Example 13:

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

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

/* Windowing Procedures */

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

/* end of “window.h” */


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

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

void windowdispatch();

main()
{
        SVCXPRT *transp;

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

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

        pmap_unset(WINDOWPROG, WINDOWVERS);

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

(continued)

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

        svc_run();       /* never returns */

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

}


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

        switch (rqstp->rq_proc) {

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

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

                /* tell the caller they made an error */

                        svcerr_decode(transp);
                        break;
                }

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

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

(continued)

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

        case RENDERSTRING_BATCHED:

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

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

                        break;
                }

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

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

                break;

        default:
                svcerr_noproc(transp);
                return;

        }       /* end switch */

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

        svc_freeargs(transp, xdr_wrapstring, &s);

}
```

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

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

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

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

Example 14:

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

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

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

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

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

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

        total_timeout.tv_sec = 20;
```

(continued)

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

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

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

        clnt_destroy(client);
}
```

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

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

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

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

Broadcast RPC 2.5.2

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

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

- Normal RPC expects one answer; broadcast RPC expects many answers (one or more answers from each responding machine).
- Only packet-oriented (connectionless) transport protocols such as UDP/IP can support broadcast RPC.
- The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if a version mismatch exists between the broadcaster and a remote service, the user of broadcast RPC never knows.
- All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible through the broadcast RPC mechanism.
- Broadcast request sizes are limited to the maximum transmission unit (mtu) of the local network.

The following is a synopsis of broadcast RPC:

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

The `eachresult()` routine is called each time a valid result is obtained. It returns the following Boolean, which indicates whether the client wants more responses:

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

If done is TRUE, broadcasting stops, and clnt_broadcast returns successfully; otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses return, the routine returns with RPC_TIMEDOUT. To interpret clnt_stat errors, feed the error code to clnt_perrno.

Authentication [3]

The RPC protocol includes a slot for authentication parameters on every call. The type of authentication used by the server and client determines the contents of the authentication parameters. A server can support the following types of authentication at once:

- AUTH_DES passes encrypted time-stamp information, allowing the client and server to perform mutual verification and authentication.
- AUTH_KERB passes encrypted Kerberos service ticket information, allowing the client and server to perform mutual verification and authentication.
- AUTH_NULL passes no authentication information (this is called *null authentication* and is the default).
- AUTH_SHORT is a shorthand form of passing UNICOS style credentials.
- AUTH_UNIX passes the UNICOS user ID, group ID, and group lists with each call.

Authentication types are fully described in appendix E, page 165.

The RPC package on the server authenticates every RPC, and, similarly, the RPC client package generates and sends authentication parameters. The authentication subsystem of the RPC package is open-ended; that is, numerous types of authentication are easy to support. This section covers UNICOS, Data Encryption Standard (DES), and Kerberos authentication.

Authentication of caller to service and vice versa are provided through call and reply messages. The call message has two authentication fields: credentials and verifier. The reply message has one authentication field, the response verifier.

The RPC protocol specification uses the XDR language described in appendix C, page 121. This protocol defines the authentication structure to be the following opaque type:

```
enum auth_flavor {
    AUTH_NULL    = 0,
    AUTH_UNIX    = 1,
    AUTH_SHORT   = 2,
    AUTH_DES     = 3,
    AUTH_KERB    = 4
    /* and more to be defined */
};

struct opaque_auth {
    union switch (enum auth_flavor) {
        default: string auth_body<00>;
    };
};
```

Any opaque_auth structure is an auth_flavor enumeration, followed by a counted string whose bytes are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields are specified by individual, independent authentication protocol specifications.

If authentication parameters are rejected, the response message contains information stating why they were rejected.

Setting up authentication 3.1

This subsection describes the requirements for null, UNICOS, DES, and Kerberos authentication.

Null authentication requirements 3.1.1

Often, calls must be made in which the client does not have to verify the identity of the server, and the server does not have to know the identity of the client. In this case, the auth_flavor value (the discriminant of the opaque_auth's union) of the RPC message's credentials, verifier, and response verifier is AUTH_NULL. The bytes of the auth_body string are undefined. The string length should be 0. Null authentication is the default; when it is used, the server accepts and performs all service requests.

UNICOS authentication requirements 3.1.2

Sometimes a server might want to limit its services to a restricted set of users. One way of doing this is by using UNICOS authentication. When UNICOS authentication is used, the `auth_flavor` discriminant of the `opaque_auth` structure has the value `AUTH_UNIX`. The bytes of the `auth_body` can then be interpreted as an `authunix_parms` structure, as defined in appendix E, page 165. In addition to the user ID, other information, including a time stamp, a machine name, the user's group ID, and a list of groups to which the user belongs, is sent to the server. The server can use the data passed in the `authunix_parms` structure any way it chooses; that is, it can use any of the fields selectively to allow or disallow services.

Unfortunately, nothing prevents malicious users from writing whatever data they choose into the `authunix_parms` structure before it is sent to the server. Thus, it is very easy for a client to deceive a server into believing it is servicing either a different user or a user who has a different set of attributes.

DES authentication requirements 3.1.3

DES authentication provides stricter security than does UNICOS authentication, allowing a server to obtain a client user's identity with a very high degree of certainty. Moreover, the client user can verify the identity of the server with whom it is communicating. Although it is technologically possible to deceive even DES authentication, to do so on a local subnet requires a lot of computational resources.

DES authentication, which is sometimes called *secure RPC*, requires that the `keyserv`(8) daemon be running on both server and client machines. The administrator must have already assigned each secure RPC user a public key/secret key pair in the `publickey` database. DES users must then register themselves by using the `keyserv` process, either automatically, by logging in with `login`(1), or manually, with the `keylogin`(1) command.

> **Note:** Because the network information service (NIS) manages the `publickey` database, NIS must be configured and running on the Cray Research system for DES authentication to work. Moreover, the Cray Research system must be in the same NIS domain as any host with whom DES authentication will be used.

Kerberos authentication requirements 3.1.4

Kerberos authentication uses encrypted Kerberos service tickets to provide more security than either UNICOS or DES authentication. Kerberos authentication requires that the Kerberos Enigma security package be installed on your system and that the site have an ONC+ license. See the *ONC+ Technology for the UNICOS Operating System*, publication SG–2169, for more information about ONC+. Users must obtain a Kerberos service ticket by using the `kinit`(1) command prior to using `AUTH_KERB` flavor RPC.

Servers using `AUTH_KERB` flavor RPC must register themselves with the authentication software by using the `svc_kerb_reg` library call. See the `kerberos_rpc`(3) man page for more information.

Client authentication 3.2

Suppose a caller creates a new RPC client handle, as in the following command:

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

By default, the type of authentication to use is set to `NULL`. However, the RPC client can choose to use UNICOS, DES, or Kerberos authentication by setting `clnt->cl_auth` after creating the RPC client handle.

For UNICOS authentication, the handle would be set as follows:

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

If an authentication failure occurs, you can use the following command instead:

```
clnt->cl_auth = authunix_create(host,uid,gid,len,aup_gids);
```

For DES authentication, the handle would be set as follows:

```
clnt->cl_auth = authdes_create(servername,credlife,&server_addr,key);
```

For Kerberos authentication, the handle would be set as follows:

```
clnt->cl_auth = authkerb_seccreate(server,instance,realm,window,timehost,
     status);
```

See appendix A, page 67, for descriptions of arguments for `authdes_create` and `authunix_create`.

Server authentication
3.3

People who develop RPC services have a more difficult time dealing with authentication issues than those implementing client applications, because the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine:

```
/*
 * An RPC service request
 */
struct svc_req {
    u_long rq_prog;        /* service program number */
    u_long rq_vers;        /* service protocol vers_num */
    u_long rq_proc;        /* desired procedure number */
    struct opaque_auth
            rq_cred;            /* raw credentials from wire */
    caddr_t rq_clntcred;        /* credentials (read only) */
};
```

The `rq_cred` field is mostly opaque, except for the style of authentication credentials, as in the following:

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

The RPC package makes the following guarantee to the service dispatch routine:

- The request's `rq_cred` field is well-formed. Thus, the service implementer might inspect the request's `rq_cred.oa_flavor` field to determine which style of authentication the caller used. If `rq_cred.oa_flavor` is `AUTH_UNIX`, the pointer `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_cred.oa_flavor` is `AUTH_DES`, the pointer `rq_clntcred` could be cast to a pointer to an `authdes_cred` structure. If `rq_cred.oa_flavor` is `AUTH_KERB`, the pointer `rq_clntcred` could be cast to a pointer to an `authkerb_clnt_cred` structure. If the style is not one of the styles that the RPC package supports, the service implementer might also want to inspect the other fields of `rq_cred`.

- 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. If `rq_clntcred` is `NULL`, the service implementer might want 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.

You can extend the remote users service example so that it computes results for all users except user ID 16, as follows:

```
nuser(rqstp, tranp)
   struct svc_req *rqstp;
   SVCXPRT *transp;
{
   struct authunix_parms *unix_cred;
   unsigned long nusers;
   struct authdes_cred *des_cred;
   struct authkerb_clnt_cred *authkerb_cred;
   int uid;
   int gid;
   int gidlen;
   int gidlist[10];
   /*
    * 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”);
           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;
```

(continued)

```
case AUTH_DES:
       des_cred =
           (struct authdes_cred *) rqstp->rq_clntcred;
       if (! netname2user(des_cred->adc_fullname.name,
           &uid, &gid, &gidlen, gidlist))
       {
           fprintf(stderr, "unknown user: %s\n",
               des_cred->adc_fullname.name);
           svcerr_systemerr(transp);
           return;
       }
       break;
   case AUTH_KERB:
        authkerb_cred =
           (struct authkerb_clnt_cred  *)rqstp->rq_clntcred;
        if (!authkerb_getucred (rqstp, &uid, &gid, gidlen, gidlist)) {
               fprintf (stderr, "unknown user:%s\n",
                  authkerb_cred->akc_fullname.pname);
               svcerr_systemerr(transp);
               return;
        }
        break;
   case AUTH_NULL:
   default:
       svcerr_weakauth(transp);
       return;
   }
   switch (rqstp->rq_proc) {
   case RUSERSPROC_NUM:
       /*
        * Explicitly disallow user with UID 16
        */
       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, "can't reply to RPC call\n");
           exit(1);
       }
       return;
   default:
       svcerr_noproc(transp);
       return;
   }
}
```

You should note the following points:

- It is customary not to check the authentication parameters associated with NULLPROC (procedure number 0). This allows any user to test for the presence of the server, simply by using rpcinfo(8).
- If the authentication parameter's type is not suitable for your service, the server should call svcerr_weakauth. For example, if the client sent credentials of type AUTH_UNIX or AUTH_NULL, and the server required credentials of type AUTH_DES, the server should call svcerr_weakauth.
- The service protocol itself should return the status for access denied. In the case of the previous example, the protocol does not have such a status; therefore, the service primitive svcerr_systemerr is called instead.

The last point underscores the relationship between the RPC authentication package and the services; RPC deals only with authentication and not with access control for individual services. Each service must implement its own access control policy and reflect that policy as return statuses in its protocol.

Record-marking standard
3.4

When RPC messages are passed on top of a byte stream protocol (such as TCP/IP), you should delimit one message from another to detect and possibly recover from user protocol errors. This is called *record marking* (RM). One RPC message fits into one RM record.

A record is composed of one or more record fragments. A *record fragment* consists of a 4-byte header, followed by 0 to 2^{31}–1 bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values: a Boolean that indicates whether the fragment is the last fragment of the record (bit value 1 implies that the fragment is the last fragment) and a 31-bit unsigned binary value that is the number of bytes in the fragment's data. The Boolean value is the high-order bit of the header; the length is the 31 low-order bits.

Note: This record specification is not in XDR standard form.

RPC Message Protocol [4]

This section defines the RPC message protocol in the External Data Representation (XDR) language.

Call and reply
4.1

The protocol begins with a call and reply, as follows:

```
enum msg type {
   CALL = 0,
   REPLY = 1
};
```

A reply to a call message indicates that the message was either accepted or rejected, as follows:

```
enum reply_stat {
MSG_ACCEPTED = 0,
MSG_DENIED = 1
};
```

If the call message was accepted, the status of an attempt to call a remote procedure is as follows:

```
enum accept_stat {
SUCCESS = 0,   /* RPC executed successfully */
PROG_UNAVAIL = 1, /* remote hasn't exported program */
PROG_MISMATCH= 2, /* remote can't support version # */
PROC_UNAVAIL = 3, /* program can't support procedure */
GARBAGE_ARGS = 4, /* procedure can't decode params */
SYSTEM_ERR = 5 /* failure in RPC system */
}
```

The following list gives reasons for a call message rejection:

```
enum reject_stat {
RPC_MISMATCH = 0,   /* RPC version number != 2 */
AUTH_ERROR = 1      /* Remote cannot authenticate caller */
};
```

The following list gives reasons for authentication failure:

```
enum auth_stat {
AUTH_BADCRED = 1,       /* Bad credentials (seal broken) */
AUTH_REJECTEDCRED=2,    /* Client must begin new session */
AUTH_BADVERF = 3,       /* Bad verifier (seal broken) */
AUTH_REJECTEDVERF=4,    /* Verifier expired or replayed */
AUTH_TOOWEAK = 5,       /* Rejected for security reasons */
AUTH_INVALIDRESP = 6,   /* Rejected because of bad response verifier */
AUTH_FAILED = 7         /* Failed for other (unknown) reason */
};
```

Message structure
4.2

All messages start with a transaction identifier, `xid`, followed by a two-armed, discriminated union. The union's discriminant is a `msg_type` that switches to one of the two types of the message. The `xid` of a `REPLY` message always matches that of the initiating `CALL` message.

> **Note:** The `xid` field is used only for clients that match reply messages with call messages; the service side cannot treat this ID as any type of sequence number.

Consider the following structure:

```
struct rpc_msg {
unsigned    xid;
union switch (enum msg_type) {
    CALL:  struct call_body;
    REPLY: struct reply_body;
} };
```

The following structure shows the body of an RPC request call. In version 2 of the RPC protocol specification, `rpcvers` must be equal to 2. The `prog`, `vers`, and `proc` fields specify the remote program, its version number, and the procedure to be called from within the remote program, respectively. These fields are followed by two authentication parameters: `cred` (authentication credentials) and `verf` (authentication verifier). The two authentication parameters are followed by the parameters to the remote procedure, which are specified by the specific program protocol.

```
struct call_body {
unsigned rpcvers;    /* Must be equal to two (2) */
unsigned prog;
unsigned vers;
unsigned proc;
struct opaque_auth cred;
struct opaque_auth verf;
/* Procedure-specific parameters start here */
};
```

The following structure shows the body of a reply to an RPC request. The call message was either accepted or rejected.

```
struct reply_body {
union switch (enum reply_stat) {
    MSG_ACCEPTED:  struct accepted_reply;
    MSG_DENIED:    struct rejected_reply;
}; };
```

The following structure shows a reply to an RPC request that the server accepted. (An error might exist, however, even though the request was accepted.) The first field is an authentication verifier that the server generates to validate itself to the caller. It is followed by a union whose discriminant is an enumeration of `accept_stat`. The `SUCCESS` arm of the union is protocol-specific. The `PROG_UNAVAIL`, `PROC_UNAVAIL`, and `GARBAGE_ARGS` arms of the union are void. The `PROG_MISMATCH` arm specifies the lowest and highest version numbers of the remote program that the server supports.

```
struct accepted_reply {
struct opaque_auth   verf;
union switch (enum accept_stat) {
    SUCCESS: struct {
        /*
         * Procedure-specific results start here
         */
```

```
    };
    PROG_MISMATCH: struct {
        unsigned low;
        unsigned high;
    };
    default: struct {
        /*
         * void.  Cases include PROG_UNAVAIL,
         * PROC_UNAVAIL, and GARBAGE_ARGS.
         */
    }; }; };
```

The following structure shows a reply to an RPC request that the server rejected. A request can be rejected either because the server is not running a compatible version of the RPC protocol (RPC_MISMATCH) or the server refuses to authenticate the caller (AUTH_ERROR). In the case of an RPC version mismatch, the server returns the lowest and highest supported RPC version numbers. In the case of refused authentication, failure status is returned.

```
struct rejected_reply {
union switch (enum reject_stat) {
    RPC_MISMATCH: struct {
        unsigned low;
        unsigned high;
    };
    AUTH_ERROR: enum auth_stat;
}; };
```

Synopsis of RPC and XDR Routines [A]

This appendix summarizes the entry points into the RPC and XDR system.

This macro destroys the authentication information associated with auth. Destruction usually involves deallocation of private data structures. The use of auth is undefined after auth_destroy is called.

auth_destroy

Format:

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

authdes_create

This routine creates and returns an RPC authentication handle that contains the following DES authentication information:

Format:

```
AUTH * authdes_create(netname, window, syncaddr, deskeyp)
    char *netname
    unsigned window;
    struct sockaddr_in *syncaddr;
    des_block *deskeyp;
```

The netname parameter is the network name of the server process owner. If the server process is a root process, you can derive the name by using the following declaration and call (argument type is character pointer):

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

rhostname is the host name of the machine on which the server process (servername) is running.

NULL specifies that the local domain name will be used.

If a user runs the server process, you can derive the name by using the following declaration and call:

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

uid is the user ID of the user whose server name you are requesting.

The window parameter is the lifetime (in seconds) for the credential. You can use a credential only once within the lifetime set by this parameter. The argument type is an unsigned integer.

The syncaddr parameter is the network address of the host with which the client must synchronize. Both client and server must be using the same time. If you are sure that the client and server are already synchronized (if, for example, both client and server are running the Network Time Protocol (NTP)), you can specify this argument as NULL. The argument type is pointer to the sockaddr_in structure (sockaddr_in*).

The deskeyp parameter is the address of a DES encryption key to use for encrypting time stamps and data. NULL indicates that you should choose a random key. The ah_key field of the authentication handle contains the encryption key. The argument type is a pointer to the des_key structure (des_key*).

authkerb_seccreate

This client side routine returns an RPC authentication handle that enables the use of the Kerberos authentication system. If the authkerb_seccreate routine fails it returns NULL. For more information see the kerberos_rpc(3) man page.

Format:

```
AUTH *
authunix_seccreate(service, srv_inst, realm
window, timehost, status)
    char *service ;
    char *srv_inst;
    char *realm
    u_int window;
    char *timehost;
    int status;
```

The service parameter is the Kerberos principal name of the service to be used.

The srv_inst parameter is the instance of the service to be called.

The `window` parameter validates the client credential, with time measured in seconds. The `ntpd`(8) daemon provides this function on a Cray Research machine.

The `timehost` parameter is optional and does nothing.

The `status` parameter is also optional. If you specify `status`, it is used to return a Kerberos error status code if an error occurs.

`authnone_create`

This routine creates and returns an RPC authentication handle that passes no usable authentication information with each RCP.

Format:

```
AUTH *
authnone_create()
```

`authunix_create`

This routine creates and returns an RPC authentication handle that contains UNICOS authentication information.

Format:

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

The `host` parameter is the name of the machine on which the information was created. The `uid` parameter is the user's ID. The `gid` parameter is the user's current group ID. The `len` and `aup_gids` parameters are counted arrays of groups to which the user belongs.

`authunix_create_default`

This routine calls `authunix_create` with the default parameters.

Format:

```
AUTH *
authunix_create_default()
```

`callrpc`

This routine calls the remote procedure associated with the program number (`prognum`), version number (`versnum`), and procedure number (`procnum`) on the machine, `host`.

Format:

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

The `inproc` parameter encodes the procedure's parameters, and the `in` parameter is the address of the procedure's arguments. The `outproc` parameter decodes the procedure's results, and the `out` parameter is the address of the destination location for the results.

If it succeeds, this routine returns 0; if it fails, it returns the value of enumeration `clnt_stat`, cast to an integer. The `clnt_perrno` routine is handy for translating failure statuses into messages.

> **Note:** Calling remote procedures with this routine uses UDP/IP as a transport; see `clntudp_create`, page 74, for restrictions.

`clnt_broadcast`

This routine is like `callrpc`, except that the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls `eachresult()`, which has the following form:

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

The `out` parameter 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.

Format:

```
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;
```

If `eachresult()` returns 0, `clnt_broadcast` waits for more replies; otherwise, it returns with appropriate status.

`clnt_call` This macro calls the remote procedure `procnum` associated with the client handle, `clnt`, which is obtained with an RPC client creation routine such as `clntudp_create`.

Format:

```
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;
```

`clnt` is the client handle, `procnum` is the procedure number, `inproc` encodes the procedure's parameters, `in` is the address of the procedure's arguments, `outproc` decodes the procedure's results, `out` is the address of the destination location for the results, and `tout` is the time allowed for results to return.

`clnt_create` This routine returns a pointer to a `CLIENT` structure. It allows users to pass the host name and protocol type as parameters of type character pointer.

Format:

```
struct CLIENT *cp;

char *hostname;
unsigned int prog;
unsigned int vers;
char *protocol;
cp=clnt_create (hostname,prog,vers,protocol);
```

`clnt_destroy` This macro destroys the client's RPC handle (`clnt`). Destruction usually involves deallocation of private data structures, including `clnt` itself. Use of `clnt` is undefined after `clnt_destroy` is called. The user must close sockets associated with `clnt`.

Format:

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

`clnt_freeres`

This macro frees any data allocated by the RPC and XDR system when it decoded the results of an RPC.

Format:

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

`clnt` is the client handle, `outproc` is the XDR routine that describes the results in simple primitives, and `out` is the address of the results. If the results were successfully freed, this routine returns 1; otherwise, it returns 0.

`clnt_geterr`

This macro copies the error structure out of the client handle (`clnt`) to the structure at address `errp`.

Format:

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

`clnt_pcreateerror`

This routine prints a message to standard error that indicates why a client RPC handle could not be created. The message is prepended with string `s` and a colon. This routine is used after a `clntraw_create`, `clnttcp_create`, or `clntudp_create` call.

Format:

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

`clnt_perrno`

This routine prints a message to standard error that corresponds to the condition indicated by `stat`. This routine is used after `callrpc`.

Format:

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

`clnt_perror`

This routine prints a message to standard error that indicates the reason an RPC failed; `clnt` is the handle used to do the call. The message is prepended with string `s` and a colon. This routine is used after `clnt_call`.

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

`clntraw_create`

This routine creates a trivial RPC client for the remote program `prognum`, version `versnum`.

Format:

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

The transport used to pass messages to the service is actually a buffer within the process address space; therefore, the corresponding RPC server should reside in the same address space (see `svcraw_create`, page 82), allowing simulation of RPC and acquisition of RPC overheads, such as round-trip times, without interference from the kernel. If the procedure fails, this routine returns `NULL`.

`clnttcp_create`

This routine creates an RPC client for the remote program `prognum`, version `versnum`; the client uses TCP/IP as a transport.

Format:

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

The remote program is located at Internet address addr. If addr–>sin_port is 0, the portmapper on the host at IP address addr is set to the actual port on which the remote program is listening (the remote portmap service is consulted for this information). The sockp parameter is a socket; if it is RPC_ANYSOCK, this routine opens a new socket and sets sockp. Because the RPC that is based on TCP uses buffered I/O, you can specify the size of the send and receive buffers by using the sendsz and recvsz parameters, respectively; values of 0 indicate that suitable defaults will be chosen. If the procedure fails, this routine returns NULL.

clntudp_create

This routine creates an RPC client for the remote program prognum, version versnum; the client uses UDP/IP as a transport.

Note: On systems that limit UDP datagrams to 8 Kbytes of data, you cannot use this transport for procedures that accept large arguments or return large results.

Format:

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

The remote program is located at Internet address addr. If addr–>sin_port is 0, the portmap on the host at IP address addr is set to the actual port on which the remote program is listening (the remote portmapper service is consulted for this information). The sockp parameter is a socket; if it is RPC_ANYSOCK, this routine opens a new socket and sets sockp. The UDP transport resends the call message in intervals of wait time until a response is received or the call times out. clnt_call specifies the total time for the call to time out.

get_myaddress

This routine puts 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).

Format:

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

pmap_getmaps

This routine is a user interface to the portmap service. It returns a list of the current RPC program-to-port mappings on the host located at IP address addr. This routine can return NULL. This routine is used when using the rpcinfo(8) command with the –p option.

Format:

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

pmap_getport

This routine is a user interface to the portmap service. It returns the port number of a waiting service that supports the program at Internet address addr, with program number prognum, version versnum, and the transport protocol associated with protocol.

A return value of 0 means that the mapping does not exist or that the RPC system failed to contact the remote portmap service. In the latter case, the global variable rpc_createerr contains the RPC status.

Format:

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

pmap_rmtcall

This routine is a user interface to the portmap service.

Format:

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

This routine instructs the portmapper on the host at IP address `*addr` to make an RPC on your behalf to a procedure on that host. If the procedure succeeds, the `*portp` parameter is changed to the program's port number. The definitions of other parameters are discussed in descriptions of `callrpc`, page 69, and `clnt_call`, page 71. You should use this procedure only in conjunction with a `ping`(8) command. See also `clnt_broadcast`, page 70.

`pmap_set` This routine is a user interface to the portmap service.

Format:

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

It establishes a mapping between a program's [`prognum`, `versnum`, `protocol`] and a port (`port`) on a machine's portmap service. The value of `protocol` is most likely `IPPROTO_UDP` or `IPPROTO_TCP`. If the program succeeds, routine `svc_register` automatically returns 1; otherwise, it returns 0.

This routine is a user interface to the portmap service.

Format:

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

`pmap_unset` This routine destroys all mappings between [`prognum,versnum,*`] and ports on the machine's portmap service. If the program succeeds, this routine automatically returns 1; otherwise, it returns 0.

`registerrpc` This routine registers procedure `procname` with the RPC service package.

Note: Remote procedures registered in this form are accessed by using the UDP/IP transport; see `svcudp_create`, page 83.

Format:

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

If a request arrives for program `prognum`, version `versnum`, and procedure `procnum`, `procname` is called with a pointer to its parameters; `procname` should return a pointer to its static results. `inproc` decodes the parameters; `outproc` encodes the results. If the registration succeeds, this routine automatically returns 0; otherwise, it returns –1.

`rpc_createerr`

This routine is a global variable whose value is set by any RPC client creation routine that does not succeed. Use the `clnt_pcreateerror` routine to print the reason for the failure.

Format:

```
struct rpc_createerr  rpc_createerr;
```

`svc_destroy`

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

Format:

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

`svc_freeargs`

This macro frees any data allocated by RPC and XDR when it used `svc_getargs` to decode the arguments to a service procedure. The parameters are those used on the `svc_getargs` macro call. If the results were successfully freed, this routine returns 1; otherwise, it returns 0.

Format:

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

`svc_getargs`

This macro decodes the arguments of an RPC request associated with the RPC service transport handle (`xprt`).

Format:

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

`inproc` is the XDR routine used to decode the arguments, and `in` is the address at which the arguments will be placed. If decoding succeeds, this routine returns 1; otherwise, it returns 0.

`svc_getcaller`

This routine is the approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle (`xprt`)

Format:

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

`svc_getreq`

This routine is similar to `svc_getreqset()`, but it is limited to 64 descriptors.

This routine is similar to `svc_getreqset()`, but it is limited to 64 descriptors.

Format:

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

`rdfds` is the read file descriptors bit mask.

`svc_getreqset`

This routine is of interest only if a service implementer does not call `svc_run`, but instead implements custom asynchronous event processing. It is called when the `select`(2) system call has determined that an RPC request has arrived on some RPC sockets.

Format:

```
svc_getreqset(rdfdsetp)
    fd_set *rdfdsetp;
```

`rdfdsetp` is a pointer to the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of `rdfdsetp` have been serviced.

The global variable `svc_fdset`, which is of type `fd_set`, reflects the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the `select`(2) system call. This is of interest only if a service implementer does not call `svc_run`, but rather does his or her own asynchronous event processing. This variable is read-only (do not pass its address to `select`(2)), yet it can change after calls to `svc_getreqset` or any creation routines. Its format is as follows:

```
fd_set   svc_fdset;
```

`svc_register` This routine associates `prognum` and `versnum` with the service dispatch procedure, `dispatch()`.

Format:

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

If `protocol` is 0, the service is not registered with the portmap service. If `protocol` is a nonzero value, a mapping of [`prognum`,`versnum`,`protocol`] to `xprt->xp_port` is established with the local portmap service (generally protocol is 0, `IPPROTO_UDP`, or `IPPROTO_TCP`). `xprt` is the RPC service transport handle. The `dispatch()` procedure has the following form:

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

If `dispatch()` succeeds, the `svc_register` routine returns 1; otherwise, it returns 0.

`svc_run` This routine never returns. It waits for RPC requests to arrive, and it calls the appropriate service procedure through `svc_getreqset` when one arrives. This procedure is usually waiting for a `select`(2) system call to return.

Format:

```
svc_run()
```

`svc_sendreply` An RPC service's dispatch routine calls this routine to send the results of an RPC.

Format:

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

`xprt` is the caller's associated transport handle, `outproc` is the XDR routine that encodes the results, and `out` is the address of the results. If the procedure succeeds, this routine returns 1; otherwise, it returns 0.

`svc_unregister`

This routine removes all mapping of [`prognum`,`versnum`] to dispatch routines, and all mapping of [`prognum`,`versnum`,*] to port numbers.

Format:

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

`svcerr_auth`

A service dispatch routine that refuses to perform an RPC because of an authentication error calls this routine.

Format:

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

`xprt` is the RPC service transport handle. `why` indicates the reason the service dispatch routine is refusing to perform the RPC.

`svcerr_decode`

A service dispatch routine that cannot successfully decode its parameters calls this routine.

Format:

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

`xprt` is the RPC service transport handle. See also `svc_getargs`, page 77.

`svcerr_noproc`

A service dispatch routine that does not implement the procedure number the caller requests calls this routine.

Format:

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

`xprt` is the RPC service transport handle.

`svcerr_noprog`

This routine is called when the specified program is not registered with the RPC package.

Format:

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

`xprt` is the RPC service transport handle. Service implementers usually do not need this routine.

`svcerr_progvers`

This routine is called when the desired version of a program is not registered with the RPC package.

Format:

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

`xprt` is the RPC service transport handle. Service implementers usually do not need this routine.

`svcerr_systemerr`

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

Format:

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

`xprt` is the RPC service transport handle.

`svcerr_weakauth`

A service dispatch routine that refuses to perform an RPC because of insufficient (but correct) authentication parameters calls this routine.

Format:

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

`xprt` is the RPC service transport handle. The routine calls `svcerr_auth(xprt,AUTH_TOOWEAK)`.

`svcraw_create`

This routine creates a trivial RPC service transport to which it returns a pointer. The transport is really a buffer within the process address space; therefore, the corresponding RPC client should reside in the same address space (see `clntraw_create()`, page 73.

Format:

```
SVCXPRT *
svcraw_create()
```

This routine allows simulation of RPC and acquisition of RPC overheads (such as round-trip times), without any kernel interference. If the procedure fails, this routine returns `NULL`.

`svctcp_create`

This routine creates an RPC service transport based on TCP/IP, to which it returns a pointer.

Format:

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

The transport is associated with the socket `sock`; `sock` can be `RPC_ANYSOCK`, in which case a new socket is created. If the socket is not bound to a local TCP port, this routine binds it to an arbitrary port. Because an RPC that is based on TCP/IP uses buffered I/O, you can specify the size of the send (`send_buf_size`) and receive (`recv_buf_size`) buffers; values of 0 indicate that suitable defaults will be chosen. On completion, the `xp_sock` field of the created `SVCXPRT` structure is the transport's socket number, and the `xp_port` field of the created `SVCXPRT` structure is the transport's port number. If the procedure fails, this routine returns `NULL`.

svcudp_create

This routine creates an RPC service transport based on UDP/IP, to which it returns a pointer.

> **Note:** On systems that can hold only up to 8 Kbytes of encoded data, you cannot use this transport for procedures that accept large arguments or return large results.

Format:

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

The transport is associated with the socket `sock`; `sock` can be `RPC_ANYSOCK`, in which case a new socket is created. If the socket is not bound to a local UDP port, this routine binds it to an arbitrary port. On completion, the `xp_sock` field of the created `SVCXPRT` structure is the transport's socket number, and the `xp_port` field of the created `SVCXPRT` structure is the transport's port number. If the routine fails, it returns `NULL`.

xdr_accepted_reply

This routine is used for describing RPC messages externally. It is useful for users who want to generate messages in the RPC style without using the RPC package.

Format:

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

`xdrs` is the XDR stream, and `ar` points to the structure that contains the reply structure.

xdr_array

This routine is a filter primitive that translates between arrays and their corresponding external representations.

Format:

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

`xdrs` is the XDR stream. `arrp` is the address of the pointer to the array. `sizep` is the address of the element count of the array; this element count cannot exceed `maxsize`. The `elsize` parameter is the size (in bytes) of each of the array's elements, and `elproc` is an XDR filter that translates between the C form of the array elements and their external representation. If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_authdes_cred`

This routine serializes/deserializes an `authdes_cred` structure. The client side uses this procedure to serialize a credential structure to be passed to the server. The server side uses this procedure to deserialize an `authdes_cred` structure from a client.

Format:

```
bool_t
xdr_authdes_cred(xdrs, cred)
    XDR *xdrs;
    struct authdes_cred *cred;
```

`xdrs` is the XDR stream. `cred` is the `authdes_cred` structure.

`xdr_authdes_verf`

This routine serializes/deserializes an `authdes_verf` structure. The client side uses it to deserialize a verification structure from the server. The server may use the routine to serialize a verification structure to be passed to the client.

Format:

```
bool_t
xdr_authdes_verf(xdrs, verf)
    register XDR *xdrs;
    register struct authdes_verf *verf;
```

`xdrs` is the XDR stream. `verf` points to a DES authentication verifier.

`xdr_authunix_parms`

This routine describes UNICOS credentials externally. It is useful for users who want to generate these credentials without using the RPC authentication package.

Format:

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

`xdrs` is the XDR stream, and `aupp` points to the structure that contains the UNICOS authentication parameters.

`xdr_bool`

This routine is a filter primitive that translates between Booleans (C integers) specified by `bp` and their external representations (`xdrs`). When encoding data, this filter produces values of either 1 or 0. If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_bytes`

This routine is a filter primitive that translates between counted byte strings and their external representations (`xdrs`).

Format:

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

`xdrs` is the XDR stream. `sp` is the address of the string pointer. The length of the string is located at address `sizep`; strings cannot be longer than `maxsize`. If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_callhdr`

This routine describes RPC headers associated with messages externally. It is useful for users who want to generate message headers in the RPC style without using the RPC package.

Format:

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

`xdrs` is the XDR stream, and `chdr` points to the structure that contains the call header data.

`xdr_callmsg`

This routine describes RPC messages externally. It is useful for users who want to generate messages in the RPC style without using the RPC package.

Format:

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

`xdrs` is the XDR stream, and `cmsg` points to the structure that contains the call message data.

`xdr_char`

This routine is a filter primitive that translates between C characters (`cp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

```
xdr_char(xdrs, cp)
    XDR *xdrs;
    char *cp;
```

This macro invokes the destroy routine associated with the XDR stream `xdrs`. Destruction usually involves freeing private data structures associated with the stream.

`xdr_destroy`

Format:

```
void
xdr_destroy(xdrs)
    XDR *xdrs;
```

Using `xdrs` after `xdr_destroy` is invoked produces undefined results.

`xdr_double`

This routine is a filter primitive that translates between C double-precision numbers (`dp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_enum`

This routine is a filter primitive that translates between C enumerations (actually integers) specified by `ep` and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_float`

This routine is a filter primitive that translates between C floating-point numbers (`fp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_getpos`

This macro invokes the get-position routine associated with the XDR stream `xdrs`.

Format:

```
u_int
xdr_getpos(xdrs)
    XDR *xdrs;
```

The routine returns an unsigned integer that indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances do not ensure this.

`xdr_inline`

This macro invokes the inline routine associated with the XDR stream `xdrs`.

Note: If the `xdr_inline` routine cannot allocate a contiguous piece of a buffer, it might return `NULL` (0). Therefore, the behavior can vary among stream instances; the routine exists for the sake of efficiency.

Format:

```
inline_t*
xdr_inline(xdrs, len)
    XDR *xdrs;
    int_len;
```

The routine returns a pointer to a contiguous piece of the stream's buffer; `len` is the byte length of the desired buffer. The pointer is cast to `inline_t*`, which is `char*` on Cray Research systems. The address returned is cast to `long *`.

`xdr_int`

This routine is a filter primitive that translates between C integers (`ip`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_long`

This routine is a filter primitive that translates between C long integers (`lp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_opaque`

This routine is a filter primitive that translates between fixed-size opaque data and its external representation (`xdrs`).

Format:

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

`cp` is the address of the opaque object; `cnt` is its size (in bytes). If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_opaque_auth`

This routine describes RPC messages externally. It is useful for users who want to generate messages in the RPC style without using the RPC package.

Format:

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

`xdrs` is the XDR stream, and `ap` points to the opaque authentication structure.

`xdr_pmap`

This routine provides an external description of parameters to various portmap procedures. It is useful for users who want to generate these parameters without using the `pmap` interface.

Format:

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

`xdrs` is the XDR stream, and `regs` points to the structure that contains registration information.

`xdr_pmaplist`

This routine describes a list of port mappings externally. It is useful for users who want to generate these parameters without using the `pmap` interface.

Format:

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

`xdrs` is the XDR stream, and `rp` is a pointer to the array that will store the `portmap` map entries.

`xdr_pointer`

This routine translates a pointer to a possibly recursive data structure. It differs from `xdr_reference` in that it can serialize and deserialize trees correctly.

Format:

```
xdr_pointer(xdrs, objpp, obj_size, xdr_obj)
    XDR *xdrs;
    char **objpp;
    u_int obj_size;
    xdrproc_t xdr_obj;
```

`xdrs` is the XDR stream, `objpp` is the address of the pointer, `obj_size` is the size of the structure to which `objpp` points, and `xdr_obj` is a pointer to a structure for each data type that is to be encoded or decoded.

`xdr_reference`

This routine is a primitive that provides pointer-dereferencing within structures.

Format:

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

`pp` is the address of the pointer, `size` is the size (in bytes) of the structure to which `*pp` points, and `proc` is an XDR procedure that filters the structure between its C form and its external representation (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_rejected_reply` This routine describes RPC reject type messages externally. It is useful for users who want to generate error messages in the RPC style without using the RPC package.

Format:

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

`xdrs` is the XDR stream, and `rr` points to the structure that contains the rejected reply information.

`xdr_replymsg` This routine describes RPC accept type messages externally. It is useful for users who want to generate messages in the RPC style without using the RPC package.

Format:

```
xdr_replymsg(xdrs, rmsg)
    XDR *xdrs;
    struct_rpc_msg *rmsg;
```

`xdrs` is the XDR stream, and `rmsg` points to the structure that contains the reply message information.

`xdr_setpos` This macro invokes the set position routine associated with the XDR stream `xdrs`.

> **Note:** It is difficult to reposition some types of XDR streams; therefore, this routine might fail with one type of stream and succeed with another.

Format:

```
xdr_setpos(xdrs, pos)
    XDR *xdrs;
    u_int pos;
```

`pos` is a position value obtained from `xdr_getpos`. If the XDR stream can be repositioned, this routine returns 1; otherwise, it returns 0.

`xdr_short`

This routine is a filter primitive that translates between C short integers (`sp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_string`

This routine is a filter primitive that translates between C strings and their corresponding external representations (`xdrs`).

Format:

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

Strings cannot be longer than `maxsize`. The `sp` parameter is the address of the string's pointer. If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdr_u_char`

This routine is a filter primitive that translates between C unsigned characters (`cp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

```
xdr_u_char(xdrs, cp)
    XDR *xdrs
    unsignedchar *cp;
```

`xdr_u_int`

This routine is a filter primitive that translates between C unsigned integers (`up`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

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

`xdr_u_long`

This routine is a filter primitive that translates between C unsigned long integers (`ulp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

```
xdr_u_long(xdrs, ulp)
    XDR *xdrs;
    unsigned_long *ulp;
```

`xdr_u_short`

This routine is a filter primitive that translates between C unsigned short integers (`usp`) and their external representations (`xdrs`). If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

```
xdr_u_short(xdrs, usp)
    XDR *xdrs;
    unsigned_short *usp;
```

`xdr_union`

This routine is a filter primitive that translates between a discriminated C union and its corresponding external representation (`xdrs`).

Format:

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

The dscmp parameter is the address of the union's discriminant, and unp is the address of the union. If the routine succeeds, it returns 1; otherwise, it returns 0. choices points to an array of xdr_discrim structures. This array must be terminated with an entry that contains a NULL procedure pointer. If the discriminant does not match any entry specified in the choices list, dfault points to the default xdr routine to use. See the rpc/xdr.h file for further details.

xdr_vector

This routine is a filter primitive that translates between vectors and their corresponding external representations (xdrs).

Format:

```
bool_t
xdr_vector(xdrs, basep, nelem, elemsize,
xdr_elem)
    XDR *xdrs;
    char *basep;
    u_int nelem;
    u_int elemsize;
    xdrproc_t xdr_elem;
```

The basep parameter is a pointer to the vector. nelem is the number of elements in the vector. elemsize is the size of each element in the vector. xdr_elem is an XDR filter that translates between the vector elements' C form and their external representation (xdrs). If the routine succeeds, it returns 1; otherwise, it returns 0.

xdr_void

This routine always returns 1.

Format:

```
xdr_void()
```

xdr_wrapstring

This routine is a primitive that calls the xdr_string (xdrs,sp,MAXUNSIGNED) routine; MAXUNSIGNED is the maximum value of an unsigned 31-bit integer. xdr_wrapstring translates null-terminated strings to or from external representation.

Format:

```
bool_t
xdr_wrapstring(xdrs, cpp)
        XDR *xdrs;
        char **cpp;
```

`xdrs` is the XDR stream, and `cpp` is the address of the pointer to the string.

`xdrmem_create`

This routine initializes the XDR stream object to which `xdrs` points.

Format:

```
void
xdrmem_create(xdrs, addr, size, op)
    XDR *xdrs;
    char *addr;
    u_int size;
    enum xdr_op op;
```

The stream's data is written to or read from a chunk of memory at location `addr`; the memory length can consist of a maximum of `size` bytes. The `op` parameter determines the direction of the XDR stream (`xdrs`); the direction can be `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`.

`xdrrec_create`

This routine initializes the XDR stream object to which `xdrs` points.

Note: This XDR stream implements an intermediate record stream. Therefore, additional bytes in the stream provide record boundary information.

Format:

```
void
xdrrec_create(xdrs, sendsize, recvsize, handle,
readit, writeit)
    XDR *xdrs;
    u_int sendsize, recvsize;
    char *handle;
    int (*readit)(), (*writeit)();
```

The stream's data is written to a buffer of size `sendsize`; a value of 0 indicates that the system should use a suitable default. The stream's data is read from a buffer of size `recvsize`; it too can be set to a suitable default by passing a 0 value. When a stream's output buffer is full, `writeit()` is called. Similarly, when a stream's input buffer is empty, `readit()` is called. The behavior of these two routines is similar to that of the UNICOS `read`(2) and `write`(2) system calls, except that `handle` is passed as the first parameter to the UNICOS routines. The caller must set the XDR stream's `op` field.

`xdrrec_endofrecord`

This routine can be invoked only on streams created by `xdrrec_create`.

Format:

```
xdrrec_endofrecord(xdrs, sendnow)
    XDR *xdrs;
    int sendnow;
```

`xdrs` is the XDR stream. The data in the output buffer is marked as a completed record; if `sendnow` is nonzero, the output buffer is optionally written out. If the routine succeeds, it returns 1; otherwise, it returns 0.

`xdrrec_eof`

This routine can be invoked only on streams (`xdrs`) created by `xdrrec_create`. This routine returns 1 if no more input is in the buffer after the rest of the current record has been consumed.

Format:

```
xdrrec_eof(xdrs)
    XDR *xdrs;
    int empty;
```

`xdrrec_skiprecord`

This routine can be invoked only on streams (`xdrs`) created by `xdrrec_create`. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. If the routine succeeds, it returns 1; otherwise, it returns 0.

Format:

```
xdrrec_skiprecord(xdrs)
    XDR *xdrs;
```

`xdrstdio_create`

This routine initializes the XDR stream object to which `xdrs` points.

> **Note:** The destroy routine associated with XDR streams calls `flush` (by using `fflush`, see `fclose`(3)) on the file stream, but never `close` (by using `close`(2)).

Format:

```
void
xdrstdio_create(xdrs, file, op)
    XDR *xdrs;
    FILE *file;
    enum xdr_op op;
```

The XDR stream data is written to or read from the stream file specified by `file`. The `op` parameter determines the direction of the XDR stream (`XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

`xprt_register`

After RPC service transport handles (`xprt`) are created, they should be registered with the RPC service package. This routine modifies the global variable `svc_fdset`. Service implementers do not usually need this routine.

Format:

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

`xprt_unregister`

Before an RPC service transport handle (`xprt`) is destroyed, it should be unregistered with the RPC service package. This routine modifies the global variable `svc_fdset`. Service implementers do not usually need this routine.

Format:

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

External Data Representation Standard: Protocol Specification [C]

This appendix contains chapter 5 of the Sun Microsystems *Network Programming* manual.

Remote Procedure Calls: Protocol Specification [D]

This appendix contains chapter 6 of the Sun Microsystems *Network Programming* manual.

This appendix contains detailed information about the data that is passed during UNICOS, Data Encryption Standard (DES), and Kerberos authentication and validation.

UNICOS authentication

E.1

The client of a remote procedure might want to identify itself as it is identified on a UNICOS system. In this case, the value of the credential's discriminant of an RPC message is AUTH_UNIX. The bytes of the credential's string encode the following (XDR) structure:

```
struct auth_unix {
    unsigned    stamp;
    string      machinename[255];
    unsigned    uid;
    unsigned    gid;
    unsigned    gids[10];
};
```

The stamp value is an arbitrary ID that the caller machine can generate. machinename is the name of the caller's machine (such as krypton). uid is the caller's effective user ID. gid is the caller's effective group ID. gids is a counted array of groups that contain the caller as a member. The verifier that accompanies the credentials should be AUTH_NULL.

The value of the discriminate of the response verifier received in the reply message from the server can be AUTH_NULL or AUTH_SHORT. In the case of AUTH_SHORT, the bytes of the response verifier's string encode an auth_opaque structure. This new auth_opaque structure can be passed to the server instead of the original AUTH_UNIX credentials. The server keeps a cache that maps shorthand auth_opaque structures (passed back by way of an AUTH_SHORT-style response verifier) to the original credentials of the caller. The caller can save network bandwidth and server CPU cycles by using the new credentials.

The server can flush the shorthand `auth_opaque` structure at any time. If this happens, the RPC message is rejected because of an authentication error. The reason for the failure is `AUTH_REJECTEDCRED`. At this point, you might want to try the original `AUTH_UNIX` credentials.

DES authentication E.2

When you use DES authentication, the authentication and validation data exchanged with each RPC call and reply become markedly more complex. When a client chooses to use DES authentication, it must make a call to `authdes_create` to build a DES authentication structure. The DES authentication structure has the following form:

```
typedef struct {
        struct  opaque_auth     ah_cred;
        struct  opaque_auth     ah_verf;
        union   des_block       ah_key;
        struct auth_ops {
                void    (*ah_nextverf)();
                int     (*ah_marshal)();        /* nextverf & serialize */
                int     (*ah_validate)();       /* validate verifier */
                int     (*ah_refresh)();        /* refresh credentials */
                void    (*ah_destroy)();        /* destroy this structure
*/
        } *ah_ops;
        caddr_t ah_private;
} AUTH;
```

The first field in the `AUTH` structure is `ah_cred`. This is the authentication handle credential field; it contains the information the client will provide the server for authentication. The second field in the `AUTH` structure is `ah_verf`. This is the authentication handle verification field; it contains the information the server will return to the client to prove its identity. Both of these fields are of type `opaque_auth`. An `opaque_auth` structure has the following form:

```
struct opaque_auth {
        enum_t  oa_flavor;              /* flavor of auth */
        caddr_t oa_base;                /* address of more auth stuff */
        u_int   oa_length;              /* not to exceed MAX_AUTH_BYTES */
};
```

The oa_flavor field specifies the type of authentication or validation being done. It can take the value AUTH_NONE, AUTH_UNIX, AUTH_SHORT, or AUTH_DES.

The oa_base field is a pointer to specific data being used for authentication or validation. In the case of AUTH_DES, the ah_cred.oa_base field is unused, but the ah_verf.oa_base field points to an area that contains an encrypted time stamp filled in by the server and checked by the client.

The oa_length field specifies the number of data bytes to which the oa_base field points.

The third field of the AUTH structure is called ah_key, the authentication handle key. This field contains a DES key used for the duration of the AUTH structure. The ah_key field is of type union des_block, which is specified as follows:

```
union des_block {
        struct {
#ifdef CRAY
                word64 both;
#else
                u_long high;
                u_long low;
#endif
        } key;
        char c[8];
};
typedef union des_block des_block;
```

The des_block is a union of 8 bytes, which constitute the DES session key. This session key exists only for the duration of the AUTH structure, which is no longer than the duration of the CLIENT structure with which this AUTH structure is associated. Typically, a new CLIENT structure is generated each time the client-side application is executed. Thus, a new DES key is generated each time the application is run. This DES session key is never sent across the network in its plain form. Instead, it is sent only after it has been encrypted as part of the ah_private data, described below.

The des_block union contains a conditional compilation statement. This is necessary because, on a Cray Research system, a long value consists of 64 bits. On most other machines that run secure RPC, a long value consists of only 32 bits. Thus, to maintain consistency, the key portion of the union is declared a structure that contains one element of type word64 on the Cray Research system. The word64 type is defined in the `<rpc/types.h>` file, and it is merely a 64-bit entity that allows the user to address the high or low 32 bits of it separately.

The `ah_key` field of the `AUTH` structure is currently used only when performing DES authentication and validation. For other types of authentication, its contents are undefined.

The next field in the `AUTH` structure is the `ah_ops` field, which is a pointer to a structure that contains function pointers specific to the authentication method. The functions pointed to are enumerated in the `AUTH` structure and include functions to get the next verifier, to marshal (generate) credentials, to validate credentials, to refresh credentials, and to destroy credentials.

The last field in the `AUTH` structure is the `ah_private` field, which is a generic pointer to data specific to the authentication method. In the case of DES authentication, the `ah_private` field points to a structure of type `ad_private`. Users should not manipulate the data within this structure. The contents of the structure are described as follows; this description is only for informational purposes.

```
/*
 * This struct is pointed to by the ah_private field of an “AUTH *”
 * when doing DES authentication.   */
struct ad_private {
        char *ad_fullname;              /* client’s full name */
        u_int ad_fullnamelen;           /* length of name, rounded up */
        char *ad_servername;            /* server’s full name */
        u_int ad_servernamelen;         /* length of name, rounded up */
        u_int ad_window;                /* client specified window */
        bool_t ad_dosync;               /* synchronize? */
        struct sockaddr ad_syncaddr;    /* remote host to synch with */
        struct timeval ad_timediff;     /* server’s time – client’s time
*/
        u_long ad_nickname;             /* server’s nickname for client */
        struct authdes_cred ad_cred;    /* storage for credential */
        struct authdes_verf ad_verf;    /* storage for verifier */
        struct timeval ad_time-stamp;   /* time-stamp sent */
        des_block ad_xkey;              /* encrypted conversation key */
};
```

This structure constitutes the authentication data that actually is sent across the network.

The first four fields of the structure are largely self-explanatory. ad_fullname and ad_servername are strings that contain the client's name and the server's name, respectively. The ad_fullnamelen and ad_servernamelen fields are the lengths of these client and server names, rounded up to a multiple of 4 bytes.

The ad_window field is an unsigned integer that contains the duration (in seconds) of the credentials. By having a small duration during which the authentication credentials are valid, the client protects itself from malicious users who might intercept these credentials and attempt to retransmit them later. If such a scheme were used, the server would detect that the credentials had expired and would deny the request.

The ad_window field is taken directly from the second parameter passed on the authdes_create call. For this reason, you should pass a relatively small number, perhaps 60, as this parameter.

The ad_dosync field is a flag that indicates whether the server and client want to synchronize their concepts of local time. Doing this ensures that the client and server agree on the end of the effective lifetime of a credential. However, synchronizing client and server is a nontrivial procedure and is not recommended. Instead, clients and servers should run an application such as ntpd(8), which implements the Network Time Protocol. By running ntpd, users are assured that the concept of current time on their local machine is essentially the same, at least for DES authentication purposes, as the current time on the server. If the third argument to the authdes_create call is not NULL, the ad_dosync field is set to TRUE.

The ad_syncaddr field is a pointer to the address of the host with whom to synchronize. This value was passed in as the third parameter of the authdes_create call. Again, you should set this parameter to NULL.

The ad_timediff field is a timeval structure, which is defined in the sys/time.h file. It contains the difference between server time and client time, and it is used as part of the synchronization mechanism.

The ad_nickname field is an unsigned long value that the client and server use to speed up validation after initial validation has completed. Essentially, the client specifies in the ad_cred field (described below) whether a "full name" or a “nickname” is being used for the credentials. When a full name is being used, the server must go through the calculations necessary to produce information that allows the client to validate confidently the server’s identity. After this is done, the client can specify that, from then on, a nickname credential can be used. This tells the server that there is no need to calculate such complex validation information for the server for each and every RPC request. It is a shorthand mechanism analogous to the `AUTH_SHORT` mechanism used with UNICOS validation.

The next field in the `ad_private` structure is the `ad_cred` field. This is an element of type `struct authdes_cred`, which is described, as follows:

```
/*
 * A DES authentication credential
 */
struct authdes_cred {
        enum authdes_namekind adc_namekind;
        struct authdes_fullname adc_fullname;
        u_long adc_nickname;
};
```

The `adc_namekind` field takes either the value `ADN_FULLNAME` or the value `ADN_NICKNAME`, depending on whether or not “real” validation is being requested.

The `adc_fullname` field, which is an `authdes_fullname` structure, looks like this:

```
/*
 * A fullname contains the network name of the client,
 * a conversation key and the window
 */
struct authdes_fullname {
        char *name;        /* network name of client, up to MAXNETNAMELEN
*/
        des_block key;     /* conversation key */
        u_long window;     /* associated window */
};
```

The types of all fields that have this structure have already been defined.

The last field in the `authdes_cred` field is `adc_nickname`, which is just an integer that the client uses to conveniently identify the server.

The ad_verf field of the ad_private structure is of type struct authdes_verf, which is described, as follows:

```
/*
 * A des authentication verifier
 */
struct authdes_verf {
        union {
                struct timeval adv_ctime;       /* clear time */
                des_block adv_xtime;            /* crypt time */
        } adv_time_u;
        u_long adv_int_u;
};
```

This is the structure that the server returns to the client to prove its identity. The first field is a union of a timeval structure and a des_block structure, both of which contain 8 bytes. It is convenient for the server to declare the structure this way, because it must encrypt a time stamp and an integer as part of the proof of identity it sends to the client. The adv_int_u long field is the integer the server encrypts.

The ad_time-stamp field of the ad_private structure is simply the time at which the client created the credential. The server uses this to detect old credentials structures.

The last field of the ad_private structure is the ad_xkey field, which is the encrypted conversation key generated by the client and sent to the server. A pointer to the plain conversation key may be passed as the fourth argument to the authdes_create call. If this pointer is NULL, authdes_create generates and encrypts a pseudo-random conversation key for the client.

Kerberos authentication

E.3

When you use Kerberos authentication, the authentication and validation data exchanged with each RPC call is similar to that used in DES authentication. An RPC client using Kerberos authentication must make a call to authkerb_seccreate to build a Kerberos authentication structure. The Kerberos authentication structure has the following form:

```
typedef struct_auth {
        struct  opaque_auth     ah_cred;
        struct  opaque_auth     ah_verf;
        union   des_block       ah_key;
```

```
        struct auth_ops {
                void    (*ah_nextverf)();
                int     (*ah_marshal)();        /* nextverf & serialize */
                int     (*ah_validate)();       /* validate verifier */
                int     (*ah_refresh)();        /* refresh credentials */
              void    (*ah_destroy)();        /* destroy this structure */
        } *ah_ops;
        caddr_t ah_private;
} AUTH;
```

The first field in the AUTH structure is ah_cred. This field points to information the client sends the server to perform authentication. The authentication data is stored in an authkerb_cred structure.

The second field in the AUTH structure, also of struct opaque_auth, is the ah_verf field. This field points to information the client sends to the server for verification. The verification data is stored in an authkerb_verf structure. Both of these fields, ah_cred and ah_verf, are of type opaque_auth.

An opaque_auth structure has the following form:

```
struct opaque_auth {
        enum_t  oa_flavor;              /* flavor of auth */
        caddr_t oa_base;                /* address of more auth stuff */
        u_int   oa_length;              /* not to exceed MAX_AUTH_BYTES */
};
```

The oa_flavor field specifies the type of authentication or validation being done. It can take the value AUTH_NONE, AUTH_UNIX, AUTH_SHORT, AUTH_DES, or AUTH_KERB. For Kerberos RPC, the oa.flavor field is set to AUTH_KERB.

The oa_base field is a pointer to specific data being used for authentication or verification. The ah_cred.oa_base field points to an authkerb_verf structure. The authkerb_cred and authkerb_verf structures are described after the _ak_private structure.

The oa_length field specifies the number of data bytes to which the oa_base field points.

The third field of the AUTH structure is called ah_key, the authentication handle key. This field contains a Kerberos session key used for the duration of the AUTH structure. The ah_key field is of type union des_block, which is specified as follows:

```
union des_block {
        struct {
#ifdef CRAY
                word64 both;
#else
                u_long high;
                u_long low;
#endif
        } key;
        char c[8];
};
typedef union des_block des_block;
```

The des_block is a union of 8 bytes, which constitute the Kerberos session key. This session key exists only for the duration of the AUTH structure, which is no longer than the duration of the CLIENT structure with which this AUTH structure is associated. Typically, a new CLIENT structure is generated each time the client-side application is executed. Thus, a new Kerberos key is generated each time the application is run.

The des_block union contains a conditional compilation statement. This is necessary because, on a Cray Research system, a long consists of 64 bits. On most other machines that run secure RPC, a long value consists of only 32 bits. Thus, to maintain consistency, the key portion of the union is declared a structure that contains one element of type word64 on the Cray Research system. The word64 type is defined in the <rpc/types.h> file, and it is merely a 64-bit entity that allows the user to address the high or low 32 bits of it separately.

The ah_key field of the AUTH structure is used only when performing Kerberos authentication and verification.

The next field in the AUTH structure is the ah_ops field, which is a pointer to a structure that contains function pointers specific to the authentication method. The functions pointed to are enumerated in the AUTH structure and include functions to get the next verifier, to marshal (generate) credentials, to validate credentials, to refresh credentials, and to destroy credentials.

The last field in the AUTH structure is the ah_private field, which is a generic pointer to data specific to the authentication method. In the case of Kerberos authentication, the ah_private field points to a structure of type _ak_private. Users should not manipulate the data within this structure. The contents of the structure are described as follows; this description is only for informational purposes.

```
/*
 * This struct is pointed to by the ah_private field of an "AUTH *"
 * when doing Kerberos authentication.   */
struct _ak_private {
        char ak_service[ANAME_SZ];     /* service name */
        char ak_srv_inst[INST_SZ];     /* server instance */
        char ak_realm[REALM_SZ];       /* realm */
        u_int ak_window;              /* client specified window */
        bool_t ak_dosync;             /* synchronize? */
        char *ak_timehost;            /* remote host to synch with */
        struct timeval ak_timediff;    /* server's time - client's time */
        u_long ak_nickname;           /* server's nickname for client */
        struct timeval ak_time-stamp;  /* time-stamp sent */
        struct authkerb_cred ak_cred;  /* storage for credential */
        struct authkerb_verf ak_verf;  /* storage for verifier */
        KTEXT_ST ak_ticket;           /* Kerberos ticket */
};
```

This structure contains additional data sent to the RPC server.

The ak_service, ak_srv_inst, ak_realm, and ak_window fields are set by the client side call authkerb_seccreate, and are assigned from the service, instance, realm, and window parameters. The ak_timehost field is always left blank. The ak_nickname field is assigned when a reply is received from the RPC server. The server returns the nickname. The nickname is used to speed up validation after the initial validation has completed.

The ak_window field is an unsigned integer that contains the duration (in seconds) of the credentials. By having a small duration during which the authentication credentials are valid, the client protects itself from malicious users who might intercept these credentials and attempt to retransmit them later. If such a scheme were used, the server would detect that the credentials had expired and would deny the request.

The ak_timediff field is a timeval structure, which is defined in the sys/time.h file. It contains the difference between server time and client time, and it is used as part of the synchronization mechanism.

The ak_nickname field is an unsigned long that the client and server use to speed up validation after initial validation has completed. Essentially, the client specifies in the ad_cred field (described below) whether a "full name" or a "nickname" is being used for the credentials. When a full name is being used, the server must go through the calculations necessary to produce information that allows the client to validate confidently the server's identity. After this is done, the client can specify that, from then on, a nickname credential can be used. This tells the server that a less complex verification and authentication may be used.

The next field in the _ak_private structure is the ak_cred field. This is an element of type struct authkerb_cred, which is described, as follows:

```
struct authkerb_cred {
        enum authkerb_namekind akc_namekind;

        struct authkerb_fullname akc_fullname;
        u_long akc_nickname;
};
```

The authkerb_namekind field takes either the value AKN_FULLNAME or the value AKN_NICKNAME, depending on whether or not full validation is being requested. When an AKN_FULLNAME value is used, an authkerb_fullname structure is sent to the server.

The akn_fullname field, which is an authkerb_fullname structure, looks like this:

```
struct authkerb_fullname {
        KTEXT_ST ticket;
        u_long window;                       /* associated window */
        };
```

The KTEXT_ST structure is a Kerberos ticket structure. The u_long window parameter is the window for the ticket. See the include file <krb/krb.h> for a description of the ticket.

Kerberos RPC places restrictions on client and server clocks. They must be synchronized within five minutes of each other. Cray Research recommends that a site run the Network Time Protocol (NTP) time protocol on the client and server to ensure synchronization.

The `AUTH_KERB` authentication flavor uses Cipher Block Chaining (CBC) mode encryption when sending a fullname credential that includes the ticket and the window. Electronic Code Book (ECB) encryption is used for nickname credentials. The Kerberos session key is used for the initial input vector for CBC encryption.

The `ak_verf` field of the `_ak_private` structure is of type `struct authkerb_verf`, which is described, as follows:

```
struct authkerb_verf {
        union {
               struct timeval akv_ctime;       /* clear time */
               des_block akv_xtime;           /* crypt time */
        } akv_time_u;
        u_long akv_int_u;
};
```

This is the structure that the server returns to the client to prove its identity. The first field is a union of a `timeval` structure and a `des_block` structure, both of which contain 8 bytes. It is convenient for the server to declare the structure this way, because it must encrypt a time stamp and an integer as part of the proof of identity it sends to the client. The `akv_int_u long` field is the integer the server encrypts.

The `ak_time-stamp` field of the `_ak_private` structure is simply the time at which the client created the credential. The server uses this to detect old credentials structures.

The server returns the nickname to the client by reusing the `authkerb_verf` structure on the return call. The client stores the nickname in its `_ak_private` structure.

Service Library Routines [F]

This appendix contains man pages for the RPC service library routines.

Index

A

Access control, 61
Array freeing, 25
Array processing.IN , 22
auth_destroy(3RPC), 67
AUTH_KERB authentication, 56
authdes_create(3RPC), 67
Authentication
  AUTH_KERB flavor, 56
  client, 56
  DES, 55, 67
  failure, 64
  handle, 68, 69
  information destroyed, 67
  Kerberos, 56
  null requirements, 54
  of caller to service, 53
  parameters, 53
  server, 57
  setting up, 54
  UNICOS, 69
  UNICOS requirements, 55
authkerb_secreate, 171
authnone_create(3RPC), 68, 69
authunix_create(3RPC), 69

B

Batching, 45
Broadcast nets, 70
Broadcast protocols, 50
Building blocks, prefabricated, 21

C

Call message, 63, 65
Callback processing, 39
callrpc(3RPC), 69
Client
  authentication, 56
  call and reply, 14
  creating handle, 56
  error routine, 16
  service library routines, 18
  source code, 20, 32
clnt_broadcast(3RPC), 70
clnt_call(3RPC), 71
credentials, 165

D

Data
  deserialized, 2
  serialized, 2
  translation, 2
DES authentication, 55
Deserializing, 2, 21

E

Entry points to RPC/XDR, 67
Environments, operating system, 2
Error messages, 9
Error routine diagnosis, 16
Exit to service loop, 14
External Data Representation (XDR), 2, 21

H

Handle setting
  DES, 56
  Kerberos, 56
  UNICOS, 56

I

Intermediate layer
  calling and replying, 20
  registering, 19

K

Kerberos authentication, 56, 171
`keyserv` process, 55

L

Layer
    highest, 17
    intermediate, 18
    lowest, 26
Layers of interface, 16
Low–level features, accessing, 33
Lowest layer
    calling, 30
    registering, 26

M

Memory allocation for XDR, 24
Message protocol, 63
Message structure, 3, 64

N

Network Programming manual, 121, 143
Null authentication, 54
`nusers` service call, 30

P

Pointer dereferencing, 23
Portmapper registering, 14
Prefabricated, building blocks, 21
Procedure number, 8
Programming examples, 11
Protocol
    call and reply, 63
    message, 63
Protocol specification registration, 4
Public key, 55

R

Record fragment, 61
Record marking, 61
Registered programs, 4
Remote procedure number, 8
Remote program number, 3
Reply to request, 65
Request
    accepted, 65
    rejected, 66
Request call, 65
Return codes, 21
Routine
    client, 16
    error, 16
    `gettransient`, 42
    service dispatch, 57
    to process array, 22
Routines
    built–in, 21
    change defaults, 26
    customer applications, 4
    for debugging, 4
    for memory allocation, 24
    library, 2
    predefined XDR, 2
    service, 18
    synopsis, 67
    user–developed, 3
    XDR, 2, 21
RPC layers, 16, 26
RPC message protocol, 63
RPC message structure, 3
RPC paradigm, 2
RPC protocol
    for batching, 45
    for broadcasting, 50
RPC request call, 65
RPC service library routines, 177
RPC services, 4
`rpcgen` utility, 2
`rpcgen`*`Programming Guide`*, 97

S

Secure RPC, 55
Select processing, 33

Semantics, 8
Serializing, 2, 21
Server
  authentication, 57
  function, 12
  source code, 19, 28
  testing for presence, 61
Service library routines, 18
Service loop exit, 14
Source code, client, 32
Status for access denied, 61
Storage allocating and freeing, 24

T

TCP processing, 34
Transports and semantics, 8

U

UNICOS authentication, 55, 165

V

Version number, 6

X

XDR memory allocation, 24
XDR routines, 2