ONC+ RPC Developer's Guide

ONC+ RPC Developer's Guide

Part No: E54804

Copyright © 1996, 2016, Oracle and/or its affiliates. All rights reserved.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Accessibilité de la documentation

Pour plus d'informations sur l'engagement d'Oracle pour l'accessibilité à la documentation, visitez le site Web Oracle Accessibility Program, à l'adresse http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info ou le site http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs si vous êtes malentendant.

Contents

Using This Documentation

- **Overview** – The *ONC+ RPC Developer's Guide* describes the programming interfaces to remote procedure call (RPC) which belongs to the ONC+® distributed services.
- **Audience** – The guide assists you in converting an existing single-computer application to a networked, distributed application, or developing and implementing distributed applications.
- **Required knowledge** – Readers of this guide should be familiar with the C programming language, and a working familiarity with the UNIX operating system. Previous experience in network programming is helpful, but is not required to use this manual.

Product Documentation Library

Documentation and resources for this product and related products are available at `http://www.oracle.com/pls/topic/lookup?ctx=E53394`.

Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

◆ ◆ ◆ **C H A P T E R   1**

Introduction to ONC+ Technologies

This chapter briefly introduces ONC+ technologies, the Oracle open systems distributed computing environment. The ONC+ technologies are the core services available to developers who implement distributed applications in a heterogeneous distributed computing environment. ONC+ technologies also include tools to administer client/server networks.

Introduction

Figure 1–1 shows an integrated view of how client-server applications are built on top of ONC+ technologies, and how they sit on top of the low-level networking protocols.

FIGURE   1   ONC+ Distributed Computing Platform
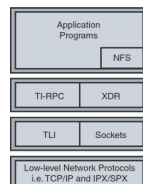
Brief Description of ONC+ Technologies

ONC+ technologies consist of a family of technologies, services, and tools. These technologies are backward compatible and interoperate with the installed base of ONC services. The main components are described. This guide covers the technologies that require the use of programming facilities.

TI-RPC

The transport-independent remote procedure call (TI-RPC) was developed as part of UNIX System V Release 4 (SVR4). TI-RPC makes RPC applications transport-independent by enabling a single binary version of a distributed program to run on multiple transports. Previously, with transport-specific RPC, the transport was bound at compile time so that applications could not use other transports unless the program was rebuilt. With TI-RPC, applications can use new transports if the system administrator updates the network configuration file and restarts the program. Thus, no changes are required to the binary application.

XDR

External data representation (XDR) is an architecture-independent specification for representing data. It resolves the differences in data byte ordering, data type size, representation, and alignment between different architectures. Applications that use XDR can exchange data across heterogeneous physical machines.

NFS

NFS is a distributed computing file system that provides transparent access to remote file systems on heterogeneous networks. In this way, users can share files among PCs, workstations, mainframes, and supercomputers. As long as users are connected to the same network, the files appear as though they are on the user's desktop. The NFS environment features Kerberos V5 authentication, multithreading, the network lock manager, and the automounter.

NFS does not have programming facilities, so it is not covered in this guide. However, the specification for NFS is available at the following sites :

- RFC 3530: NFS Version 4 Protocol Specification – `http://tools.ietf.org/html/rfc3530`
- RFC 1094: NFS: Network File System Protocol Specification – `http://tools.ietf.org/html/rfc1094`
- RFC 1813: NFS Version 3 Protocol Specification – `http://tools.ietf.org/html/rfc1813`

2

Introduction to TI-RPC

This section provides an overview of TI-RPC . The information presented is most useful to someone new to RPC. See the "Glossary" on page 277 for the definition of the terms used in this guide.

Topics covered in this chapter include:

- "What Is TI-RPC?" on page 17
- "TI-RPC Issues" on page 18
- "Overview of Interface Routines" on page 20
- "Network Selection" on page 22
- "Transport Selection" on page 23
- "Address Look-up Services" on page 24

What Is TI-RPC?

TI-RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local, procedure calling so that the called procedure need not exist in the same address space as the calling procedure. The two processes might be on the same system, or they might be on different systems with a network connecting them.

By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and enables the application to use a variety of transports.

FIGURE   2   How RPC Works
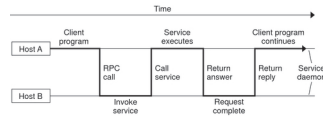
An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure.

Figure 2–1 shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or the request times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues.

RPC specifically supports network applications. TI-RPC runs on available networking mechanisms such as TCP/IP. Other RPC standards are OSF DCE (based on Apollo's NCS system), Xerox Courier, and Netwise.

TI-RPC Issues

A number of issues help to characterize a particular RPC implementation.

- How are parameters and results passed?
- How is binding carried out?
- How are transport protocols dealt with?
- What are the call semantics?
- What data representation is used?

Parameter Passing

TI-RPC allows a single parameter to be passed from client to server. If more than one parameter is required, the components can be combined into a structure that is counted as a single element. Information passed from server to client is passed as the function's return value. Information cannot be passed back from server to client through the parameter list.

Binding

The client must know how to contact the service. The two necessary aspects are finding out which host the server is on, and then connecting to the actual server process. On each host, a service called `rpcbind` manages RPC services. TI-RPC uses the available host-naming services, such as the `hosts` file and DNS, to locate a host.

Transport Protocol

The transport protocol specifies how the call message and the reply message are transmitted between client and server. TS-RPC used TCP and UDP as transport protocols, but the current version of TI-RPC is transport independent, so it works with any transport protocol.

Call Semantics

Call semantics define what the client can assume about the execution of the remote procedure; in particular, how many times the procedure was executed. These semantics are important in dealing with error conditions. The three alternatives are *exactly once*, *at most once*, and *at least once*. ONC+ provides *at least once* semantics. Procedures called remotely are *idempotent*: they should return the same result each time they are called, even through several iterations.

Data Representation

Data representation describes the format used for parameters and results as they are passed between processes. To function on a variety of system architectures, RPC requires a standard data representation. TI-RPC uses external data representation (XDR). XDR is a system-independent data description and encoding protocol. Using XDR, RPC can handle arbitrary data structures, regardless of the byte orders or structure layout conventions of the different hosts. For a detailed discussion of XDR, see Appendix A, "XDR Technical Note" and Appendix C, "XDR Protocol Specification".

Program, Version, and Procedure Numbers

A remote procedure is uniquely identified by the triple:

- Program number
- Version number
- Procedure number

The *program* number identifies a group of related remote procedures, each of which has a unique procedure number.

A program can consist of one or more *versions*. Each version consists of a collection of procedures that are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously.

Each version contains a number of procedures that can be called remotely. Each procedure has a *procedure* number.

"Program and Procedure Numbers" on page 205 lists the range of values and their significance and tells you how to have a program number assigned to your RPC program. A list of mappings of RPC service name to program number is available in the RPC network database /etc/rpc.

Overview of Interface Routines

RPC has multiple levels of application interface to its services. These levels provide different degrees of control balanced with different amounts of interface code to implement, in order of increasing control and complexity. This section gives a summary of the routines available at each level.

Simplified Interface Routines

The simplified interfaces are used to make remote procedure calls to routines on other systems, and specify only the type of transport to use. The routines at this level are used for most applications. Descriptions and code samples are in the section "Simplified Interface" on page 61.

TABLE 1   RPC Routines–Simplified Level

Routine	Function
rpc_reg()	Registers a procedure as an RPC program on all transports of the specified type
rpc_call()	Remotely calls the specified procedure on the specified remote host
rpc_broadcast()	Broadcasts a call message across all transports of the specified type

Standard Interface Routines

The standard interfaces are divided into *top level*, *intermediate level*, *expert level*, and *bottom level*. These interfaces give a programmer much greater control over communication

parameters such as the transport being used, how long to wait before responding to errors and retransmitting requests, and so on.

Top-Level Routines

At the top level, the interface is still simple, but the program has to create a client handle before making a call or create a server handle before receiving calls. If you want the application to run on all transports, use this interface. You can find the use of these routines and code samples in "Top-Level Interface" on page 69 .

TABLE 2   RPC Routines–Top Level

Routine	Description
clnt_create()	Generic client creation. The program tells clnt_create() where the server is located and the type of transport to use.
clnt_create_timed()	Similar to clnt_create() but enables the programmer to specify the maximum time allowed for each type of transport tried during the creation attempt.
svc_create()	Creates server handles for all transports of the specified type. The program tells svc_create() which dispatch function to use.
clnt_call()()	Client calls a procedure to send a request to the server.

Intermediate-Level Routines

The intermediate level interface of RPC enables you to you control details. Programs written at these lower levels are more complicated but run more efficiently. The intermediate level enables you to specify the transport to use. "Intermediate-Level Interface" on page 73 describes the use of these routines and code samples.

TABLE 3   RPC Routines–Intermediate Level

Routine	Description
clnt_tp_create()	Creates a client handle for the specified transport
clnt_tp_create_timed()	Similar to clnt_tp_create() but enables the programmer to specify the maximum time allowed
svc_tp_create()	Creates a server handle for the specified transport
clnt_call()()	Client calls a procedure to send a request to the server

Expert-Level Routines

The expert level contains a larger set of routines with which to specify transport-related parameters. "Expert-Level Interface" on page 76 describes the use of these routines and code samples.

TABLE 4   RPC Routines–Expert Level

Routine	Description
`clnt_tli_create()`	Creates a client handle for the specified transport
`svc_tli_create()`	Creates a server handle for the specified transport
`rpcb_set()`	Calls `rpcbind` to set a map between an RPC service and a network address
`rpcb_unset()`	Deletes a mapping set by `rpcb_set()`
`rpcb_getaddr()`	Calls `rpcbind()` to get the transport addresses of specified RPC services
`svc_reg()`	Associates the specified program and version number pair with the specified dispatch routine
`svc_unreg()`	Deletes an association set by `svc_reg()`
`clnt_call()()`	Client calls a procedure to send a request to the server

Bottom-Level Routines

The bottom level contains routines used for full control of transport options. "Bottom-Level Interface" on page 80 describes these routines.

TABLE 5   RPC Routines–Bottom Level

Routine	Description
`clnt_dg_create()`	Creates an RPC client handle for the specified remote program using a connectionless transport
`svc_dg_create()`	Creates an RPC server handle using a connectionless transport
`clnt_vc_create()`	Creates an RPC client handle for the specified remote program using a connection-oriented transport
`svc_vc_create()`	Creates an RPC server handle using a connection-oriented transport
`clnt_call()()`	Client calls a procedure to send a request to the server

Network Selection

You can write programs to run on a specific transport or transport type, or to operate on a system-chosen or user-chosen transport. Two mechanisms for network selection are the `/etc/netconfig` database and the environmental variable `NETPATH`. These mechanisms enable a fine degree of control over network selection: a user can specify a preferred transport and an application will use it if it can. If the specified transport is inappropriate, the application automatically tries other transports with the right characteristics.

`/etc/netconfig` lists the transports available to the host and identifies them by type. `NETPATH` is optional and enables you to specify a transport or selection of transports from the list in `/etc/netconfig`. By setting the `NETPATH`, you specify the order in which the application tries the

available transports. If `NETPATH` is not set, the system defaults to all visible transports specified in `/etc/netconfig`, in the order that they appear in that file.

For more details on network selection, see the `getnetconfig(3NSL)` and `netconfig(4)` man pages.

RPC divides selectable transports into the types described in the following table.

TABLE 6   `nettype` Parameters

Value	Meaning
NULL	Same as selecting `netpath`.
visible	Uses the transports chosen with the visible flag ('v') set in their `/etc/netconfig` entries.
circuit_v	Same as `visible`, but restricted to connection-oriented transports. Transports are selected in the order listed in `/etc/netconfig`.
datagram_v	Same as `visible`, but restricted to connectionless transports.
circuit_n	Uses the connection-oriented transports chosen in the order defined in `NETPATH`.
datagram_n	Uses the connectionless transports chosen in the order defined in `NETPATH`.
udp	Specifies Internet User Datagram Protocol (UDP).
tcp	Specifies Internet Transport Control Protocol (TCP).

Transport Selection

RPC services are supported on both circuit-oriented and datagram transports. The selection of the transport depends on the requirements of the application.

Choose a datagram transport if the application has all of the following characteristics:

- Calls to the procedures do not change the state of the procedure or of associated data.
- The size of both the arguments and results is smaller than the transport packet size.
- The server is required to handle hundreds of clients. A datagram server does not keep any state data on clients, so it can potentially handle many clients. A circuit-oriented server keeps state data on each open client connection, so the number of clients is limited by the system resources.

Choose a circuit-oriented transport if the application has any of the following characteristics:

- The application can tolerate or justify the higher cost of connection setup compared to datagram transports.
- Calls to the procedures can change the state of the procedure or of associated data.
- The size of either the arguments or the results exceeds the maximum size of a datagram packet.

Name-to-Address Translation

Each transport has an associated set of routines that translate between universal network addresses (string representations of transport addresses) and the local address representation. These universal addresses are passed around within the RPC system (for example, between rpcbind and a client). A runtime linkable library that contains the name-to-address translation routines is associated with each transport. Table 7, "Name-to-Address Translation Routines," on page 24 shows the main translation routines.

For more details on these routines, see the netdir(3NSL) man page. Note that the netconfig structure in each case provides the context for name-to-address translations.

TABLE 7   Name-to-Address Translation Routines

`netdir_getbyname()`	Translates from host or service pairs (for example `server1`, `rpcbind`) and a `netconfig` structure to a set of `netbuf` addresses. `netbufs` are Transport Level Interface (TLI) structures that contain transport-specific addresses at runtime.
	Translates from `netbuf()` addresses and a `netconfig` structure to host or service pairs.
`uaddr2taddr()`	Translates from universal addresses and a `netconfig()` structure to `netbuf` addresses.
`taddr2uaddr()`	Translates from `netbuf` addresses and a `netconfig` structure to universal addresses.

Address Look-up Services

Transport services do not provide address look-up services. They provide only message transfer across a network. A client program needs a way to obtain the address of its server program. In previous system releases this service was performed by portmap. In this release, rpcbind replaces the portmap utility.

RPC makes no assumption about the structure of a network address. It handles universal addresses specified only as null-terminated strings of ASCII characters. RPC translates universal addresses into local transport addresses by using routines specific to the transport. For more details on these routines, see the netdir(3NSL) and rpcbind(3NSL) man pages.

rpcbind enables you to perform the following operations:

- Add a registration
- Delete a registration
- Get address of a specified program number, version number, and transport
- Get the complete registration list
- Perform a remote call for a client

Registering Addresses

rpcbind maps RPC services to their addresses, so rpcbind's address must be known. The name-to-address translation routines must reserve a known address for each type of transport used. For example, in the Internet domain, rpcbind has port number 111 on both TCP and UDP. When rpcbind is started, it registers its location on each of the transports supported by the host. rpcbind is the only RPC service that must have a known address.

For each supported transport, rpcbind registers the addresses of RPC services and makes the addresses available to clients. A service makes its address available to clients by registering the address with the rpcbind daemon. The address of the service is then available to rpcinfo(1M) and to programs using library routines named in the rpcbind(3NSL) man page. No client or server can assume the network address of an RPC service.

Client and server programs and client and server hosts are usually distinct but they need not be. A server program can also be a client program. When one server calls another rpcbind server it makes the call as a client.

To find a remote program's address, a client sends an RPC message to a host's rpcbind daemon. If the service is on the host, the daemon returns the address in an RPC reply message. The client program can then send RPC messages to the server's address. A client program can minimize its calls to rpcbind by storing the network addresses of recently called remote programs.

The RPCBPROC_CALLIT procedure of rpcbind enables a client to make a remote procedure call without knowing the address of the server. The client passes the target procedure's program number, version number, procedure number, and calling arguments in an RPC call message. rpcbind looks up the target procedure's address in the address map and sends an RPC call message, including the arguments received from the client, to the target procedure.

When the target procedure returns results, RPCBPROC_CALLIT passes them to the client program. It also returns the target procedure's universal address so that the client can later call it directly.

The RPC library provides an interface to all rpcbind procedures. Some of the RPC library procedures also call rpcbind automatically for client and server programs. For details, see "RPC Language Specification" on page 222.

Reporting RPC Information

rpcinfo is a utility that reports current RPC information registered with rpcbind. rpcinfo, with either rpcbind or the portmap utility, reports the universal addresses and the transports for all registered RPC services on a specified host. rpcinfo can call a specific version of a specific program on a specific host and report whether a response is received. rpcinfo can also delete registrations. For details, see the rpcinfo(1M) man page.

3

rpcgen Programming Guide

This section introduces the `rpcgen` tool and provides a tutorial with code examples and usage of the available compile-time flags. See "Glossary" on page 277 for the definition of the terms used in this chapter.

The topics covered in this chapter include:

What Is `rpcgen`?

The `rpcgen` tool generates remote program interface modules. It compiles source code written in the RPC language. The RPC language is similar in syntax and structure to C. The `rpcgen` tool produces one or more C language source modules, which are then compiled by a C compiler.

The default output of `rpcgen` is:

- A header file of definitions common to the server and the client
- A set of `XDR` routines that translate each data type defined in the header file
- A stub program for the server
- A stub program for the client

`rpcgen` can optionally generate:

- Various transports
- A timeout for servers
- Server stubs that are multithread safe
- Server stubs that are not `main` programs
- C-style arguments passing ANSI C-compliant code
- An RPC dispatch table that checks authorizations and invokes service routines

rpcgen significantly reduces the development time that would otherwise be spent developing low-level routines. Handwritten routines link easily with the rpcgen output. For a discussion of RPC programming without rpcgen, see Chapter 4, "Programmer's Interface to RPC".

Software Environment Features

This section lists the features found in the current rpcgen code generator.

- Template Generation: rpcgen generates client-side, server-side, and makefile templates. See "Compile-Time Client and Server Templates" on page 41 for the list of options.
- C-style Mode: rpcgen has two compilation modes, C-style and default. In C-style mode arguments can be passed by value, instead of as pointers to a structure. It also supports passing multiple arguments. The default mode is the same as in previous releases. See "Compile-Time C-style Mode" on page 42 for the example code for both modes.
- Multithread-Safe Code: rpcgen generates MT-safe code for use in a threaded environment. By default, the code generated by rpcgen is not MT safe. See "Compile-Time MT-Safe Code" on page 45 for the description and example code.
- Multithread Auto Mode: rpcgen generates MT-safe server stubs that operate in the MT Auto mode. See "Compile-Time MT Auto Mode" on page 50 for the definition and example code.
- Library Selection: rpcgen uses library calls for either TS-RPC or TI-RPC. See "Compile-Time TI-RPC or TS-RPC Library Selection" on page 51.
- ANSI C-compliant Code: The output generated by rpcgen conforms to ANSI C standards. See "Compile-Time ANSI C-compliant Code" on page 51.

rpcgen Tutorial

rpcgen provides programmers a direct way to write distributed applications. Server procedures can be written in any language that observes procedure-calling conventions. These procedures are linked with the server stub produced by rpcgen to form an executable server program. Client procedures are written and linked in the same way.

This section presents some basic rpcgen programming examples. Refer also to the rpcgen(1) man page.

Converting Local Procedures to Remote Procedures

Assume that an application runs on a single computer and you want to convert it to run in a "distributed" manner on a network. This example shows the stepwise conversion of this program that writes a message to the system console. The following code example shows the original program.

EXAMPLE 1   Single Process Version of `printmesg.c`

```
/* printmsg.c: print a message on the console */

#include <stdio.h>

main(argc, argv)
 int argc;
 char *argv[];
{
 char *message;

 if (argc != 2) {
fprintf(stderr, "usage: %s <message>\n",
     argv[0]);
  exit(1);
 }
 message = argv[1];
 if (!printmessage(message)) {
  fprintf(stderr,"%s: couldn't print your
     message\n",argv[0]);
 exit(1);
 }
 printf("Message Delivered!\n");
 exit(0);
}
 /* Print a message to the console.
 * Return a boolean indicating whether
 * the message was actually printed. */
 printmessage(msg)
 char *msg;
{
 FILE *f;
  f = fopen("/dev/console", "w");
 if (f == (FILE *)NULL) {
  return (0);
 }
 fprintf(f, "%s\n", msg);
 fclose(f);
 return(1);}
```

For local use on a single system, this program could be compiled and executed as follows:

```
$ cc printmsg.c -o printmsg
$ printmsg "Hello, there."
Message delivered!
$
```

If the `printmessage()` function is turned into a remote procedure, the function can be called from anywhere in the network.

First, determine the data types of all procedure-calling arguments and the resulting argument. The calling argument of `printmessage()` is a string, and the result is an integer. You can write a protocol specification in the RPC language that describes the remote version of `printmessage()`. The RPC language source code for such a specification is:

```
/* msg.x: Remote msg printing protocol */
 program MESSAGEPROG {
     version PRINTMESSAGEVERS {
         int PRINTMESSAGE(string) = 1;
    } = 1;
} = 0x20000001;
```

Remote procedures are always declared as part of remote programs. The previous code declares an entire remote program that contains the single procedure `PRINTMESSAGE`. In this example, the `PRINTMESSAGE` procedure is declared to be procedure `1`, in version `1` of the remote program `MESSAGEPROG`, with the program number `0x20000001`. See Appendix B, "RPC Protocol and Language Specification" for guidance on choosing program numbers.

Version numbers are incremented when functionality is changed in the remote program. Existing procedures can be changed or new ones can be added. More than one version of a remote program can be defined and a version can have more than one procedure defined.

Note that the program and procedure names are declared with all capital letters.

Note also that the argument type is `string` and not `char *` as it would be in C. This is because a `char *` in C is ambiguous. `char` usually means an array of characters, but it could also represent a pointer to a single character. In the RPC language, a null-terminated array of `char` is called a `string`.

You have just two more programs to write:

- The remote procedure itself
- The main client program that calls it

Example 2, "RPC Version of `printmsg.c`," on page 31 is a remote procedure that implements the `PRINTMESSAGE` procedure in Example 1, "Single Process Version of `printmesg.c`," on page 29.

EXAMPLE 2   RPC Version of `printmsg.c`

```
/*
 * msg_proc.c: implementation of the
 * remote procedure "printmessage"
 */
#include <stdio.h>
#include "msg.h"     /* msg.h generated by rpcgen */

int *
printmessage_1(msg, req)
 char **msg;
 struct svc_req *req;                     /* details of call */
{
 static int result;   /* must be static! */
 FILE *f;

 f = fopen("/dev/console", "w");
 if (f == (FILE *)NULL) {
  result = 0;
  return (&result);
 }
 fprintf(f, "%s\n", *msg);
 fclose(f);
 result = 1;
 return (&result);}
```

The declaration of the remote procedure `printmessage_1()` differs from that of the local procedure `printmessage()` in four ways:

1. `printmessage_1()` takes a pointer to the character array instead of the pointer itself. This principle is true of all remote procedures when the `-N` option is not used. These procedures always take pointers to their arguments rather than the arguments themselves. Without the `-N` option, remote procedures are always called with a single argument. If more than one argument is required the arguments must be passed in a `struct`.

2. `printmessage_1()` is called with two arguments. The second argument contains information on the context of an invocation: the program, version, and procedure numbers; raw and canonical credentials; and an SVCXPRT structure pointer. The SVCXPRT structure contains transport information. All of the information is made available in case the invoked procedure requires it to perform the request.

3. `printmessage_1()` returns a pointer to an integer instead of the integer itself. This principle is also true of remote procedures when the `-N` option is not used. These procedures return pointers to the result. The result should be declared `static` unless the `-M` (multithread) or `-A` (Auto mode) options are used. Ordinarily, if the result is declared local to the remote procedure, references to the result by the server stub are invalid after the remote procedure returns. In the case of `-M` and `-A` options, a pointer to the result is passed as a third argument to the procedure, so the result is not declared in the procedure.

4. An _1 is appended to the printmessage_1() name. In general, all remote procedures calls generated by rpcgen are named as follows: the procedure name in the program definition (here PRINTMESSAGE) is converted to all lowercase letters, an underbar (_) is appended to it, and the version number (here 1) is appended. This naming scheme enables you to have multiple versions of the same procedure.

The following code example shows the main client program that calls the remote procedure.

EXAMPLE 3   Client Program to Call printmsg.c

```
/*
 * rprintmsg.c: remote version
 * of "printmsg.c"
 */
#include <stdio.h>
#include "msg.h"   /* msg.h generated by rpcgen */

main(argc, argv)
 int argc;
 char *argv[];
{
 CLIENT *clnt;
 int *result;
 char *server;
 char *message;

 if (argc != 3) {
  fprintf(stderr, "usage: %s host
     message\n", argv[0]);
  exit(1);
 }

 server = argv[1];
 message = argv[2];
 /*
  * Create client "handle" used for
 * calling MESSAGEPROG on the server
  * designated on the command line.
  */
 clnt = clnt_create(server, MESSAGEPROG,
        PRINTMESSAGEVERS,
        "visible");
 if (clnt == (CLIENT *)NULL) {
  /*
   * Couldn't establish connection
    * with server.
   * Print error message and die.
   */
  clnt_pcreateerror(server);
  exit(1);
```

```
}
 /*
 * Call the remote procedure
* "printmessage" on the server
 */
result = printmessage_1(&message, clnt);
if (result == (int *)NULL) {
 /*
  * An error occurred while calling
   * the server.
  * Print error message and die.
  */
 clnt_perror(clnt, server);
 exit(1);
}
/* Okay, we successfully called
* the remote procedure.
*/
if (*result == 0) {
 /*
  * Server was unable to print
   * our message.
  * Print error message and die.
  */
 fprintf(stderr,
 "%s: could not print your message\n",argv[0]);
 exit(1);
}
 /* The message got printed on the
* server's console
*/
printf("Message delivered to %s\n",
   server);
clnt_destroy( clnt );
exit(0);}
```

In the example code, a client handle is created by the RPC library routine clnt_create().
This client handle is passed to the stub routine that calls the remote procedure. See Chapter 4,
"Programmer's Interface to RPC" for details on how the client handle can be created in
other ways. If no more calls are to be made using the client handle, destroy it with a call to
clnt_destroy() to conserve system resources.

The last parameter to clnt_create() is visible, which specifies that any transport noted as
visible in /etc/netconfig can be used. For further information on transports, see the /etc/
netconfig file and its description in *Oracle Solaris 11.3 Programming Interfaces Guide*.

The remote procedure printmessage_1() is called exactly the same way as it is declared in
msg_proc.c, except for the inserted client handle as the second argument. The remote procedure
also returns a pointer to the result instead of the result.

The remote procedure call can fail in two ways. The RPC mechanism can fail or an error can occur in the execution of the remote procedure. In the former case, the remote procedure printmessage_1() returns a NULL. In the latter case, the error reporting is application dependent. Here, the error is returned through *result.*

The compile commands for the printmsg example are:

```
$ rpcgen msg.x
$ cc rprintmsg.c msg_clnt.c -o rprintmsg -lnsl
$ cc msg_proc.c msg_svc.c -o msg_server -lnsl
```

rpcgen is used to generate the header files (msg.h), client stub (msg_clnt.c), and server stub (msg_svc.c). Then, two programs are compiled: the client program rprintmsg and the server program msg_server. The C object files must be linked with the library libnsl, which contains all of the networking functions, including those for RPC and XDR.

In this example, no XDR routines were generated because the application uses only the basic types that are included in libnsl.

rpcgen received the input file msg.x and created:

- A header file called msg.h that contained #define statements for MESSAGEPROG, MESSAGEVERS, and PRINTMESSAGE for use in the other modules. This file must be included by both the client and server modules.
- The client stub routines in the msg_clnt.c file. Only one routine, the printmessage_1() routine, was called from the rprintmsg client program. If the name of an rpcgen input file is FOO.x, the client stub's output file is called FOO_clnt.c.
- The server program in msg_svc.c that calls printmessage_1() from msg_proc.c. The rule for naming the server output file is similar to that of the client: for an input file called FOO.x, the output server file is named FOO_svc.c.

After the server program is created, it is installed on a remote system and run. If the systems are homogeneous, the server binary can just be copied. If they are not homogeneous, the server source files must be copied to and compiled on the remote system. For this example, the remote system is called remote and the local system is called local. The server is started from the shell on the remote system:

```
remote$ msg_server
```

Server processes generated with rpcgen always run in the background. You do not have to follow the server's invocation with an ampersand (&). Servers generated by rpcgen can also be invoked by port monitors like listen() and inetd(), instead of from the command line.

Thereafter, a user on local can print a message on the console of system remote as follows:

```
local$ rprintmsg remote "Hello, there."
```

Using rprintmsg, a user can print a message on any system console, including the local console, when the server msg_server is running on the target system.

Passing Complex Data Structures

shows how to generate client and server RPC code. rpcgen can also be used to generate XDR routines, which are the routines that convert local data structures into XDR format and the reverse.

The following code example presents a complete RPC service: a remote directory listing service, built using rpcgen to generate both stub routines and the XDR routines.

EXAMPLE 4   RPC Protocol Description File: dir.x

```
/*
 * dir.x: Remote directory listing protocol
 *
 * This example demonstrates the functions of rpcgen.
 */

const MAXNAMELEN = 255;                   /* max length of directory entry */
typedef string nametype<MAXNAMELEN>; /* director entry */
typedef struct namenode *namelist;      /* link in the listing */

/* A node in the directory listing */
struct namenode {
    nametype name; /* name of directory entry */
    namelist next; /* next entry */
};
/*
 * The result of a READDIR operation
 *
 * A truly portable application would use an agreed upon list of
 * error codes rather than (as this sample program does) rely upon
 * passing UNIX errno's back.
 *
 * In this example: The union is used here to discriminate between
 * successful and unsuccessful remote calls.
 */
union readdir_res switch (int errno) {
    case 0:
        namelist list; /* no error: return directory listing */
    default:
        void;          /* error occurred: nothing else to return */
};
/* The directory program definition */
program DIRPROG {
    version DIRVERS {
        readdir_res
        READDIR(nametype) = 1;
    } = 1;
} = 0x20000076;
```

You can redefine types (like `readdir_res` in the previous example) using the `struct`, `union`, and `enum` RPC language keywords. These keywords are not used in later declarations of variables of those types. For example, if you define a union, `foo`, you declare using only `foo`, and not `union foo`.

`rpcgen` compiles RPC unions into C structures. Do not declare C unions using the `union` keyword.

Running `rpcgen` on `dir.x` generates four output files:

- Header file
- Client stub
- Server skeleton
- XDR routines in the file `dir_xdr.c`.

The `dir_xdr.c` file contains the XDR routines to convert declared data types from the host platform representation into XDR format, and the reverse.

For each RPC data type used in the `.x` file, `rpcgen` assumes that `libnsl` contains a routine with a name that is the name of the data type, prepended by the XDR routine header `xdr_` (for example, `xdr_int`). If a data type is defined in the `.x` file, `rpcgen` generates the required `xdr_` routine. If there is no data type definition in the `.x` source file (for example, `msg.x`), then no `_xdr.c` file is generated.

You can write a `.x` source file that uses a data type not supported by `libnsl`, and deliberately omit defining the type in the `.x` file. In doing so, you must provide the `xdr_` routine. This is a way to provide your own customized `xdr_` routines. See Chapter 4, "Programmer's Interface to RPC" for more details on passing arbitrary data types. The server-side of the `READDIR` procedure is shown in the following example.

EXAMPLE 5   Server `dir_proc.c` Example

```
/*
 * dir_proc.c: remote readdir
 * implementation
 */

#include <dirent.h>
#include "dir.h"             /* Created by rpcgen */

extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(dirname, req)
    nametype *dirname;
```

```
        struct svc_req *req;
{
    DIR *dirp;
    struct dirent *d;
    namelist nl;
    namelist *nlp;
    static readdir_res res; /* must be static! */

    /* Open directory */
    dirp = opendir(*dirname);
    if (dirp == (DIR *)NULL) {
        res.errno = errno;
        return (&res);
    }
    /* Free previous result */
    xdr_free(xdr_readdir_res, &res);
    /*
     * Collect directory entries.
     * Memory allocated here is free by xdr_free the next time
     * readdir_1 is called
     */
    nlp = &res.readdir_res_u.list;
    while (d = readdir(dirp)) {
        nl = *nlp = (namenode *)
        malloc(sizeof(namenode));
        if (nl == (namenode *) NULL) {
            res.errno = EAGAIN;
            closedir(dirp);
            return(&res);
        }
        nl->name = strdup(d->d_name);
        nlp = &nl->next;
    }
    *nlp = (namelist)NULL;
    /* Return the result */
    res.errno = 0;
    closedir(dirp);
    return (&res);
}
```

The following code example shows the client-side implementation of the READDIR procedure.

EXAMPLE 6   Client-side Implementation of rls.c

```
/*
 * rls.c: Remote directory listing client
 */

#include <stdio.h>
#include "dir.h"                        /* generated by rpcgen */
```

```
extern int errno;

main(argc, argv)
 int argc;
 char *argv[];
{
 CLIENT *clnt;
 char *server;
 char *dir;
 readdir_res *result;
 namelist nl;
  if (argc != 3) {
  fprintf(stderr, "usage: %s host directory\n",argv[0]);
  exit(1);
 }
 server = argv[1];
 dir = argv[2];
 /*
  * Create client "handle" used for
 * calling MESSAGEPROG on the server
 * designated on the command line.
  */
 cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
 if (clnt == (CLIENT *)NULL) {
  clnt_pcreateerror(server);
  exit(1);
 }
 result = readdir_1(&dir, clnt);
 if (result == (readdir_res *)NULL) {
  clnt_perror(clnt, server);
  exit(1);
 }
 /* Okay, we successfully called
 * the remote procedure.
 */
 if (result->errno != 0) {
  /* Remote system error. Print
    * error message and die.
    */
  errno = result->errno;
  perror(dir);
  exit(1);
 }
 /* Successfully got a directory listing.
 * Print it.
 */
 for (nl = result->readdir_res_u.list;
  nl != NULL;
  nl = nl->next) {
  printf("%s\n", nl->name);
```

```
 }
 xdr_free(xdr_readdir_res, result);
 clnt_destroy(cl);
 exit(0);}
```

As in other examples, execution is on systems named local and remote. The files are compiled and run as follows:

```
 remote$ rpcgen dir.x
 remote$ cc -c dir_xdr.c
 remote$ cc rls.c dir_clnt.c dir_xdr.o -o rls -lnsl
 remote$ cc dir_svc.c dir_proc.c dir_xdr.o -o dir_svc -lnsl
 remote$ dir_svc
```

When you install rls() on system local, you can list the contents of /usr/share/lib on system remote as follows:

```
 local$ rls remote /usr/share/lib
 ascii
 eqnchar
 greek
 kbd
 marg8
 tabclr
 tabs
 tabs4
 local
$
```

Client code generated by rpcgen does not release the memory allocated for the results of the RPC call. Call xdr_free() to release the memory when you are finished with it. Calling xdr_free() is similar to calling the free() routine, except that you pass the XDR routine for the result. In this example, after printing the list, xdr_free(xdr_readdir_res, result); was called.

Note - Use xdr_free() to release memory allocated by malloc(). Failure to use xdr_free() to release memory results in memory leaks.

Preprocessing Directives

rpcgen supports C and other preprocessing features. C preprocessing is performed on rpcgen input files before they are compiled. All standard C preprocessing directives are allowed in the.x source files. Depending on the type of output file being generated, five symbols are defined by rpcgen.

rpcgen provides an additional preprocessing feature: any line that begins with a percent sign (%) is passed directly to the output file, with no action on the line's content. Use caution

because `rpcgen` does not always place the lines where you intend. Check the output source file and, if needed, edit it.

`rpcgen` uses the preprocessing directives listed in the following table.

TABLE 8   rpcgen Preprocessing Directives

Symbol	Use
RPC_HDR	Header file output
RPC_XDR	XDR routine output
RPC_SVC	Server stub output
RPC_CLNT	Client stub output
RPC_TBL	Index table output

The following code example is a simple `rpcgen` example. Note the use of `rpcgen`'s pre-processing features.

EXAMPLE 7   Time Protocol rpcgen Source

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
 version TIMEVERS {
  unsigned int TIMEGET() = 1;
 } = 1;
} = 0x20000044;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
% static int thetime;
%
% thetime = time(0);
% return (&thetime);
%}
#endif
```

cpp Directive

`rpcgen` supports C preprocessing features. `rpcgen` defaults to use `/usr/ccs/lib/cpp` as the C preprocessor. If that fails, `rpcgen` tries to use `/lib/cpp`. You can specify a library containing a different `cpp` to `rpcgen` with the `-Y` flag.

For example, if `/usr/local/bin/cpp` exists, you can specify it to `rpcgen` as follows:

```
rpcgen -Y /usr/local/bin test.x
```

Compile-Time Flags

This section describes the `rpcgen` options available at compile time. The following table summarizes the options that are discussed in this section.

TABLE 9   `rpcgen` Compile-Time Flags

Option	Flag	Comments
Templates	-a, -Sc, -Ss, -Sm	See Table 10, "rpcgen Template Selection Flags," on page 41
C-style	-N	Also called Newstyle mode
ANSI C	-C	Often used with the -N option
MT-safe code	-M	For use in multithreaded environments
MT auto mode	-A	-A also turns on -M option
TS-RPC library	-b	TI-RPC library is default
`xdr_inline` count	-i	Uses five packed elements as default, but other number can be specified

Compile-Time Client and Server Templates

`rpcgen` generates sample code for the client and server sides. Use the options described in the following table to generate the desired templates.

TABLE 10   `rpcgen` Template Selection Flags

Flag	Function
-a	Generates all template files
-Sc	Generates client-side template
-Ss	Generates server-side template
-Sm	Generates makefile template

The files can be used as guides by filling in the missing parts. These files are in addition to the stubs generated.

A C-style mode server template is generated from the `add.x` source by the command `rpcgen -N -Ss -o add_server_template.c add.x`

The result is stored in the file `add_server_template.c`. A C-style mode, client template for the same `add.x` source is generated with the command `rpcgen -N -Sc -o add_client_template.c add.x`

The result is stored in the file `add_client_template.c`. A makefile template for the same `add.x` source is generated with the command `rpcgen -N -Sm -o mkfile_template add.x`

The result is stored in the file `mkfile_template`. It can be used to compile the client and the server. The `-a` flag, used in the command `rpcgen -N -a add.x`, generates all three template files. The client template is stored in `add_client.c`, the server template in `add_server.c`, and the makefile template in`makefile.a`. If any of these files already exists, `rpcgen` displays an error message and exits.

Note - When you generate template files, give them new names to avoid the files being overwritten the next time `rpcgen` is executed.

Compile-Time C-style Mode

Also called Newstyle mode, the `-N` flag causes `rpcgen` to produce code in which arguments are passed by value and multiple arguments are passed without a `struct`. These changes enable RPC code that is more like C and other high-level languages. For compatibility with existing programs and make files, the previous standard mode of argument passing is the default. The following examples demonstrate the new feature. The source modules for both modes, C-style and default, are shown in Example 8, "C-style Mode Version of `add.x`," on page 42 and Example 9, "Default Mode Version of `add.x`," on page 42 respectively.

EXAMPLE 8   C-style Mode Version of `add.x`

```
/*
 * This program contains a procedure
 * to add 2 numbers. It demonstrates
 * the C-style mode argument passing.
 * Note that add() has 2 arguments.
 */
program ADDPROG {    /* program number */
 version ADDVER {   /* version number */
  int add(int, int) = 1;  /* procedure */
 } = 1;
} = 0x20000199;
```

EXAMPLE 9   Default Mode Version of `add.x`

```
/*
 * This program contains a procedure
```

```
 * to add 2 numbers. It demonstrates
 * the "default" mode argument passing.
 * In this mode rpcgen can process
 * only one argument.
 */
struct add_arg {
 int first;
 int second;
};
program ADDPROG {    /* program number */
 version ADDVER {   /* version number */
  int add (add_arg) = 1;  /* procedure */
 } = 1;
} = 0x20000199;
```

The next four examples show the resulting client-side templates.

EXAMPLE 10   C-style Mode Client Stub for `add.x`

```
/*
 * The C-style client side main
 * routine calls the  add() function
 * on the remote rpc server
 */
#include <stdio.h>
#include "add.h"

main(argc, argv)
int argc;
char *argv[];
{
 CLIENT *clnt;
 int *result,x,y;

 if(argc != 4) {
  printf("usage: %s host num1 num2\n" argv[0]);
  exit(1);
 }
 /* create client handle -
 * bind to server
 */
 clnt = clnt_create(argv[1], ADDPROG, ADDVER, "udp");
 if (clnt == NULL) {
  clnt_pcreateerror(argv[1]);
  exit(1);
 }
 x = atoi(argv[2]);
 y = atoi(argv[3]);
 /*
  * invoke remote procedure: Note that
```

```
 * multiple arguments can be passed to
 * add_l() instead of a pointer
 */
result = add_1(x, y, clnt);
if (result == (int *) NULL) {
 clnt_perror(clnt, "call failed:");
 exit(1);
} else {
 printf("Success: %d + %d = %d\n", x, y, *result);
}
exit(0);
}
```

The following code example shows how the default mode code differs from C-style mode code.

EXAMPLE 11   Default Mode Client

```
arg.first = atoi(argv[2]);
arg.second = atoi(argv[3]);
/*
 * invoke remote procedure -- note
        * that a pointer to the argument has
 * to be passed to the client stub
 */
result = add_1(&arg, clnt);
```

The server-side procedure in C-style mode is shown in the following example.

EXAMPLE 12   C-style Mode Server

```
#include "add.h"

int *
add_1(arg1, arg2, rqstp)
 int arg1;
 int arg2;
 struct svc_req *rqstp;
{
 static int result;

 result = arg1 + arg2;
 return(&result);
}
```

The server-side procedure in default mode is shown in the following code example.

EXAMPLE 13   Default Mode Server Stub

```
#include "add.h"
```

```
int *
add_1(argp, rqstp)
 add_arg *argp;
 struct svc_req *rqstp;
{
 static int result;

 result = argp->first + argp->second;
 return(&result);
}
```

Compile-Time MT-Safe Code

By default, the code generated by rpcgen is not MT safe. It uses unprotected global variables and returns results in the form of static variables. The -M flag generates MT-safe code that can be used in a multithreaded environment. This code can be used with the C-style flag, the ANSI C flag, or both.

An example of an MT-safe program with this interface follows. The rpcgen protocol file is msg.x, shown in the following code example.

EXAMPLE 14 MT-Safe Program: msg

```
program MESSAGEPROG {
version PRINTMESSAGE {
        int PRINTMESSAGE(string) = 1;
        } = 1;
} = 0x4001;
```

A string is passed to the remote procedure, which prints it and returns the length of the string to the client. The MT-safe stubs are generated with the rpcgen -M msg.x command.

Client-side code that could be used with this protocol file is shown in the following code example.

EXAMPLE 15 MT-Safe Client Stub

```
#include "msg.h"

void
messageprog_1(host)
    char *host;
{
    CLIENT *clnt;
    enum clnt_stat retval_1;
    int result_1; char * printmessage_1_arg;
```

```
            clnt = clnt_create(host, MESSAGEPROG, PRINTMESSAGE, "netpath");

            if (clnt == (CLIENT *) NULL) {
                clnt_pcreateerror(host)
                exit(1);
            }
            printmessage_1_arg = (char *) malloc(256);
            strcpy(printmessage_1_arg, "Hello World");
            retval_1 = printmessage_1(&printmessage_1_arg, &result_1,clnt);
            if (retval_1 != RPC_SUCCESS) {
                clnt_perror(clnt, "call failed");
            }
            printf("result = %d\n", result_1);
            clnt_destroy(clnt);
        }

        main(argc, argv)
            int argc;
            char *argv[];
        {
            char *host;

            if (argc < 2) {
                printf("usage: %s server_host\n", argv[0]);
                exit(1);
            }
            host = argv[1];
            messageprog_1(host);
        }
```

A pointer to both the arguments and the results needs to be passed in to the rpcgen-generated code in order to preserve re-entrancy. The value returned by the stub function indicates whether this call is a success or a failure. The stub returns RPC_SUCCESS if the call is successful. Compare the MT-safe client stub, generated with the -M option, and the MT-unsafe client stub shown in Example 16, "Client Stub (MT Unsafe)," on page 46. The client stub that is not MT-safe uses a static variable to store returned results and can use only one thread at a time.

EXAMPLE   16   Client Stub (MT Unsafe)

```
int *
printmessage_1(argp, clnt)
 char **argp;
 CLIENT *clnt;
{
 static int clnt_res;
 memset((char *)&clnt_res, 0, sizeof (clnt_res));
 if (clnt_call(clnt, PRINTMESSAGE,
  (xdrproc_t) xdr_wrapstring, (caddr_t) argp,
  (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
```

```
                        return (NULL);
 }
 return (&clnt_res);
}
```

The server side code is shown in the following example.

Note - When compiling a server that uses MT-safe mode, you must link in the threads library. To do so, specify the -lthread option in the compile command.

EXAMPLE 17   MT-Safe Server Stub

```
#include "msg.h"

#include <syslog.h>

bool_t
printmessage_1_svc(argp, result, rqstp)
 char **argp;
 int *result;
 struct svc_req *rqstp;
{
 int retval;

 if (*argp == NULL) {
  syslog(LOG_INFO, "argp is NULL\n");
  *result = 0;
 }
 else {
  syslog("argp is %s\n", *argp);
  *result = strlen (*argp);
 }
 retval = 1;
 return (retval);
}

int
messageprog_1_freeresult(transp, xdr_result, result)
 SVCXPRT *transp;
 xdrproc_t xdr_result;
 caddr_t result;
{
 /*
  * Insert additional freeing code here,
        * if needed
  */
 (void) xdr_free(xdr_result, result);
}
```

The server-side code should not use statics to store returned results. A pointer to the result is passed in and this should be used to pass the result back to the calling routine. A return value of 1 indicates success to the calling routine, while 0 indicates a failure.

In addition, the code generated by rpcgen also generates a call to a routine to free any memory that might have been allocated when the procedure was called. To prevent memory leaks, any memory allocated in the service routine needs to be freed in this routine. messageprog_1_freeresult() frees the memory.

Normally, xdr_free() frees any allocated memory for you. In this example, no memory was allocated, so no freeing needs to take place.

The following add.x file shows the use of the -M flag with the C-style and ANSI C flag.

EXAMPLE 18   MT-Safe Program: add.x

```
program ADDPROG {
 version ADDVER {
 int add(int, int) = 1;
 } = 1;
}= 199;
```

This program adds two numbers and returns its result to the client. rpcgen is invoked on it, with the rpcgen -N -M -C add.x command. The following example shows the multithreaded client code to call this code.

EXAMPLE 19   MT-Safe Client: add.x

```
/*
 * This client-side main routine starts up a number of threads,
 * each of which calls the server concurrently.
 */

#include "add.h"

CLIENT *clnt;
#define NUMCLIENTS 5
struct argrec {
 int arg1;
 int arg2;
};

/*
 * Keeps count of number of threads running
 */
int numrunning;
mutex_t numrun_lock;
cond_t condnum;
```

```
void
addprog(struct argrec *args)
{
 enum clnt_stat retval;
 int result;
 /* call server code */
 retval = add_1(args->arg1, args->arg2,
          &result, clnt);
 if (retval != RPC_SUCCESS) {
  clnt_perror(clnt, "call failed");
 } else
  printf("thread #%x call succeeded,
     result = %d\n", thr_getself(),
     result);
 /* decrement the number of running threads */
 mutex_lock(&numrun_lock);
 numrunning--;
 cond_signal(&condnum);
 mutex_unlock(&numrun_lock);
 thr_exit(NULL);
}

main(int argc, char *argv[])
{
 char *host;
 struct argrec args[NUMCLIENTS];
 int i;
 thread_t mt;
 int ret;

 if (argc < 2) {
  printf("usage: %s server_host\n", argv[0]);
  exit(1);
 }
 host = argv[1];
 clnt = clnt_create(host, ADDPROG, ADDVER, "netpath");
 if (clnt == (CLIENT *) NULL) {
  clnt_pcreateerror(host);
  exit(1);
 };
 mutex_init(&numrun_lock, USYNC_THREAD, NULL);
 cond_init(&condnum, USYNC_THREAD, NULL);
 numrunning = 0;

 /* Start up separate threads */
 for (i = 0; i < NUMCLIENTS; i++) {
  args[i].arg1 = i;
  args[i].arg2 = i + 1;
  ret = thr_create(NULL, NULL, addprog, (char *) &args[i],
       THR_NEW_LWP, &mt);
  if (ret == 0)
```

```
    numrunning++;
 }

 mutex_lock(&numrun_lock);
 /* are any threads still running ? */
 while (numrunning != 0)
  cond_wait(&condnum, &numrun_lock);
 mutex_unlock(&numrun_lock);
 clnt_destroy(clnt);}
```

The server-side procedure is shown in the following example.

Note - When compiling a server that uses MT-safe mode, you must link in the threads library.
To do so, specify the -lthread option in the compile command.

EXAMPLE 20   MT-Safe Server: add.x

```
add_1_svc(int arg1, int arg2, int *result, struct svc_req
            *rqstp)
{
 bool_t retval;
 /* Compute result */
 *result = arg1 + arg2;
 retval = 1;
 return (retval);
}

/* Routine for freeing memory that may
 * be allocated in the server procedure
 */
int
addprog_1_freeresult(SVCXPRT *transp, xdrproc_t xdr_result, caddr_t result)

{
 (void) xdr_free(xdr_result, result);
}
```

Compile-Time MT Auto Mode

MT Auto mode enables RPC servers to automatically use Oracle Solaris threads to process
client requests concurrently. Use the -A option to generate RPC code in MT Auto mode. The
-A option also has the effect of turning on the -M option, so -M does not need to be explicitly
specified. The -M option is necessary because any code that is generated has to be multithread
safe.

The section Chapter 7, "Multithreaded RPC Programming" contains further discussion on
multithreaded RPC. See also "MT Auto Mode" on page 138.

An example of an Auto mode program generated by `rpcgen` follows in the `rpcgen` protocol file `time.x`. A string is passed to the remote procedure, which prints the string and returns its length to the client.

EXAMPLE 21   MT Auto Mode: `time.x`

```
program TIMEPROG {
 version TIMEVERS {
  unsigned int TIMEGET(void) = 1;
  void TIMESET(unsigned) = 2;
 } = 1;
} = 0x20000044;
```

The MT-safe stubs are generated with the `rpcgen -A time.x` command.

When the `-A` option is used, the generated server code contains instructions for enabling MT Auto mode for the server.

Note - When compiling a server that uses MT Auto mode, you must link in the threads library. To do so, specify the `-lthread` option in the compile command.

Compile-Time TI-RPC or TS-RPC Library Selection

In older Oracle Solaris releases, `rpcgen` created stubs that used the socket functions. With the current Oracle Solaris release, you can use either the transport-independent RPC (TI-RPC) or the transport-specific socket (TS-RPC) routines. These routines provides backward compatibility with previous releases. The default uses the TI-RPC interfaces. The `-b` flag tells `rpcgen` to create TS-RPC variant source code as its output.

Compile-Time ANSI C-compliant Code

`rpcgen` can also produce output that is compatible with ANSI C. This feature is selected with the `-C` compile flag and is most often used with the `-N` flag, described in .

The `add.x` example of the server template is generated by the `rpcgen -N -C -Ss -o add_server_template.c add.x` command:

Note that on the C++ 3.0 server, remote procedure names require an `_svc` suffix. In the following example, the `add.x` template and the `-C` compile flag produce the client side `add_1` and the server stub `add_1_svc`.

EXAMPLE 22   rpcgen ANSI C Server Template

```
/*
 * This is a template. Use it to
 * develop your own functions.
 */
#include <c_varieties.h>
#include "add.h"

int *
add_1_svc(int arg1, int arg2, struct svc_req *rqstp)
{
 static int result;
 /*
  * insert server code here
  */
 return(&result);
}
```

This output conforms to the syntax requirements and structure of ANSI C. The header files that are generated when this option is invoked can be used with ANSI C or with C++.

Compile-Time `xdr_inline()` Count

rpcgen tries to generate more efficient code by using `xdr_inline()` when possible (see the `xdr_admin(3NSL)` man page). When a structure contains elements that `xdr_inline()` can be used on (for example `integer()`, `long()`, `bool()`), the relevant portion of the structure is packed with `xdr_inline()`. A default of five or more packed elements in sequence causes inline code to be generated. You can change this default with the -i flag. The `rpcgen -i 3 test.x` command causes `rpcgen` to start generating inline code after three qualifying elements are found in sequence. The `rpcgen -i 0 test.x` command prevents any inline code from being generated.

In most situations, you do not need to use the -i flag. The `_xdr.c` stub is the only file affected by this feature.

`rpcgen` Programming Techniques

This section suggests some common RPC and `rpcgen` programming techniques.

TABLE 11   RPC Programming Techniques

Technique	Description
Network type	rpcgen can produce server code for specific transport types.

Technique	Description
Define statements	You can define C-preprocessing symbols on rpcgen command lines.
Broadcast calls	Servers need not send error replies to broadcast calls.
Debugging applications	Debug as normal function calls, then change to a distributed application.
Port monitor support	Port monitors can "listen" on behalf of RPC servers.
Dispatch tables	Programs can access server dispatch tables.
Time-out changes	You can change client default time-out periods.
Authentication	Clients can authenticate themselves to servers; the appropriate servers can examine client authentication information.

Network Types/Transport Selection

rpcgen takes optional arguments that enable a programmer to specify desired network types or specific network identifiers. For details of network selection, see *Oracle Solaris 11.3 Programming Interfaces Guide*.

The -s flag creates a server that responds to requests on the specified type of transport. For example, the command rpcgen -s datagram_n prot.x writes a server to standard output that responds to any of the connectionless transports specified in the NETPATH environment variable, or in /etc/netconfig, if NETPATH is not defined. A command line can contain multiple -s flags and their network types.

Similarly, the -n flag creates a server that responds only to requests from the transport specified by a single network identifier.

Note - Be careful using servers created by rpcgen with the -n flag. Network identifiers are host specific, so the resulting server might not run as expected on other hosts.

Command-Line Define Statements

You can define C-preprocessing symbols and assign values to them from the command line. Command-line define statements can be used to generate conditional debugging code when the DEBUG symbol is defined. For example:

```
$ rpcgen -DDEBUG proto.x
```

Server Response to Broadcast Calls

When a procedure has been called through broadcast RPC and cannot provide a useful response, the server should send no reply to the client, thus reducing network traffic. To prevent

the server from replying, a remote procedure can return NULL as its result. The server code generated by rpcgen detects this return and sends no reply.

The following code example is a procedure that replies only if it reaches an NFS server.

EXAMPLE 23   NFS Server Response to Broadcast Calls

```
void *
reply_if_nfsserver()
{
 char notnull; /*only here so we can use its address */

 if( access( "/etc/dfs/sharetab", F_OK ) < 0 ) {
  /* prevent RPC from replying */
  return( (void *) NULL );
 }
 /* Assign notnull a non-null value then RPC sends a reply */
 return( (void *) &notnull );
}
```

A procedure must return a non-NULL pointer when it is appropriate for RPC library routines to send a reply.

In the example, if the procedure reply_if_nfsserver() is defined to return nonvoid values, the return value &notnull should point to a static variable.

Port Monitor Support

Port monitors such as inetd and listen can monitor network addresses for specified RPC services. When a request arrives for a particular service, the port monitor spawns a server process. After the call has been serviced, the server can exit. This technique conserves system resources. The main server function generated by rpcgen allows invocation by inetd. See "Using inetd" on page 118 for details.

Services might wait for a specified interval after completing a service request, in case another request follows. If no call arrives in the specified time, the server exits, and some port monitors, like inetd, continue to monitor for the server. If a later request for the service occurs, the port monitor gives the request to a waiting server process (if any), rather than spawning a new process.

Note - When monitoring for a server, some port monitors, like listen(), always spawn a new process in response to a service request. If a server is used with such a monitor, the server should exit immediately on completion.

By default, services created using rpcgen wait for 120 seconds after servicing a request before exiting. You can change the interval with the -K flag. In the following example, the server waits

for 20 seconds before exiting. To create a server that exits immediately, you can use zero value for the interval period.

```
rpcgen -K 20 proto.x

rpcgen -K 0 proto.x
```

To create a server that never exits, the value is -K -1.

Time-out Changes

After sending a request to the server, a client program waits for a default period (25 seconds) to receive a reply. You can change this timeout by using the clnt_control() routine. See "Standard Interfaces" on page 69 for additional uses of the clnt_control() routine. See also the rpc(3NSL) man page. When considering time-out periods, be sure to allow the minimum amount of time required for "round-trip" communications over the network. The following code example illustrates the use of clnt_control().

EXAMPLE 24   clnt_control Routine

```
struct timeval tv;
CLIENT *clnt;
clnt = clnt_create( "somehost", SOMEPROG, SOMEVERS, "visible" );

if (clnt == (CLIENT *)NULL)
 exit(1);
tv.tv_sec = 60; /* change time-out to 60 seconds */
tv.tv_usec = 0;
clnt_control(clnt, CLSET_TIMEOUT, &tv);
```

Client Authentication

The client create routines do not have any facilities for client authentication. Some clients might have to authenticate themselves to the server.

The following example illustrates one of the least secure authentication methods in common use. See "Authentication" on page 100 for information on more secure authentication techniques.

EXAMPLE 25   AUTH_SYS Authentication Program

```
CLIENT *clnt;
clnt = clnt_create( "somehost", SOMEPROG, SOMEVERS, "visible" );
if (clnt != (CLIENT *)NULL) {
```

```
 /* To set AUTH_SYS style authentication */
  clnt->cl_auth = authsys_createdefault();
}
```

Authentication information is important to servers that have to achieve some level of security. This extra information is supplied to the server as a second argument.

The following example is for a server that checks client authentication data. It is modified from `printmessage_1()` in . The code allows only superusers to print a message to the console.

EXAMPLE 26   `printmsg_1` for Superuser

```
int *
printmessage_1(msg, req)
 char **msg;
 struct svc_req  *req;
{
 static int result; /* Must be static */
 FILE *f;
 struct authsys_parms *aup;

 aup = (struct authsys_parms *)req->rq_clntcred;
 if (aup->aup_uid != 0) {
  result = 0;
  return (&result)
 }

/* Same code as before. */
}
```

Dispatch Tables

Sometimes programs should have access to the dispatch tables used by the RPC package. For example, the server dispatch routine might check authorization and then invoke the service routine. Or, a client library might handle the details of storage management and XDR data conversion.

When invoked with the -T option, `rpcgen` generates RPC dispatch tables for each program defined in the protocol description file, `proto.x`, in the file `proto_tbl.i`. The suffix `.i` stands for "index." You can invoke `rpcgen` with the -t option to build only the header file. You cannot invoke `rpcgen` in C-style mode (-N) with either the -T or -t flag.

Each entry in the dispatch table is a `struct rpcgen_table`, defined in the header file `proto.h` as follows:

```
struct rpcgen_table {
```

```
   char *(*proc)();
   xdrproc_t xdr_arg;
   unsigned len_arg;
   xdrproc_t xdr_res;
   xdrproc_t len_res
};
```

where:

proc is a pointer to the service routine

xdr_arg is a pointer to the input (argument) xdr routine

len_arg is the length in bytes of the input argument

xdr_res is a pointer to the output (result) xdr routine

len_res is the length in bytes of the output result

The table, named dirprog_1_table for the dir.x example, is indexed by procedure number. The variable *dirprog_1_nproc* contains the number of entries in the table.

The find_proc() routine shows an example of how to locate a procedure in the dispatch tables.

EXAMPLE 27   Using a Dispatch Table

```
struct rpcgen_table *
find_proc(proc)
   rpcproc_t proc;
{
   if (proc >= dirprog_1_nproc)
       /* error */
   else
       return (&dirprog_1_table[proc]);
}
```

Each entry in the dispatch table contains a pointer to the corresponding service routine. However, that service routine is usually not defined in the client code. To avoid generating unresolved external references, and to require only one source file for the dispatch table, the rpcgen service routine initializer is RPCGEN_ACTION(proc_ver).

Using this technique, the same dispatch table can be included in both the client and the server. Use the following define statement when compiling the client.

```
#define RPCGEN_ACTION(routine) 0
```

Use the following define when writing the server.

```
#define RPCGEN_ACTION(routine)routine
```

64–Bit Considerations for rpcgen

In [Example 27, "Using a Dispatch Table," on page 57](#) proc is declared as type rpcproc_t. Formerly, RPC programs, versions, procedures, and ports were declared to be of type u_long. On a 32–bit system, a u_long is a 4–byte quantity (as is an int); on a 64–bit system, a u_long is an 8-byte quantity. The data types rpcprog_t, rpcvers_t, rpc_proc_t, and rpcport_t, introduced in the Solaris 7 environment, should be used whenever possible in declaring RPC programs, versions, procedures, and ports in place of both u_long and long. These newer types provide backwards compatibility with 32–bit systems. They are guaranteed to be 4–byte quantities no matter which system rpcgen is run on. While rpcgen programs using u_long versions of programs, versions, and procedures can still run, they have different consequences on 32– and 64–bit systems. For that reason, replace them with the appropriate newer data types. In fact, avoid using long and u_long whenever possible.

Beginning with the Solaris 7 environment, source files created with rpcgen containing XDR routines use different inline macros depending on whether the code is to run on a 32–bit or 64–bit system. Specifically, the source files will use the IXDR_GET_INT32() and IXDR_PUT_INT32() macros instead of IXDR_GETLONG() and IXDR_PUTLONG(). For example, if the rpcgen source file foo.x contains the following code, the resulting foo_xdr.c file ensures that the correct inline macro is used.

```
struct foo {
        char    c;
        int     i1;
        int     i2;
        int     i3;
        long    l;
        short   s;
};

#if defined(_LP64) || defined(_KERNEL)
 register int *buf;

#else
 register long *buf;

#endif
  . . .
 #if defined(_LP64) || defined(_KERNEL)

  IXDR_PUT_INT32(buf, objp->i1);

 IXDR_PUT_INT32(buf, objp->i2);
 IXDR_PUT_INT32(buf, objp->i3);

 IXDR_PUT_INT32(buf, objp->l);
```

```
 IXDR_PUT_SHORT(buf, objp->s);
#else


 IXDR_PUT_LONG(buf, objp->i1);

 IXDR_PUT_LONG(buf, objp->i2);

 IXDR_PUT_LONG(buf, objp->i3);
 IXDR_PUT_LONG(buf, objp->l);

 IXDR_PUT_SHORT(buf, objp->s);
#endif
```

The code declares *buf* to be either `int` or `long`, depending on whether the system is 64–bit or 32–bit.

Currently, data types transported by using RPC are limited in size to 4-byte quantities (32 bits). The 8-byte `long` is provided to enable applications to make maximum use of 64–bit architecture. However, programmers should avoid using `long`s, and functions that use `long`s, such as `x_putlong()`, in favor of `int`s whenever possible. As noted previously, RPC programs, versions, procedures, and ports have their own dedicated types. `xdr_long()` fails if the data value is not between `INT32_MIN` and `INT32_MAX`. Also, the data could be truncated if inline macros such as `IXDR_GET_LONG()` and `IXDR_PUT_LONG()` are used. The same concerns apply to `u_long` variables. See also the xdr_long(3NSL) man page.

IPv6 Considerations for `rpcgen`

Only TI-RPC supports IPv6 transport. If an application is intended to run over IPv6, now or in the future, you must not use the backward compatibility switch. The selection of IPv4 or IPv6 is determined by the respective order of associated entries in `/etc/netconfig`.

Debugging Applications

To simplify the testing and debugging process, first test the client program and the server procedure in a single process by linking them with each other rather than with the client and server skeletons. Comment out calls to the client create RPC library routines (see the rpc_clnt_create(3NSL) man page) and the authentication routines. Do not link with `libnsl`.

Link the procedures from the previous example by using the command `cc rls.c dir_clnt.c dir_proc.c -o rls`

With the RPC and XDR functions commented out, the procedure calls execute as ordinary local function calls, and the program is debugged with a local debugger such as `dbxtool`. When the

program works, the client program is linked to the client skeleton produced by `rpcgen` and the
server procedures are linked to the server skeleton produced by `rpcgen`.

You can also use the Raw RPC mode to test the XDR routines. For details, see "Testing
Programs Using Low-Level Raw RPC" on page 84 for details.

Two kinds of errors can happen in an RPC call. The first kind of error is caused by a problem
with the mechanism of the remote procedure calls. Examples of this problem are:

- The procedure is not available
- The remote server is not responding
- The remote server is unable to decode the arguments.

In Example 26, "`printmsg_1` for Superuser," on page 56, an RPC error happens if `result`
is `NULL`. The reason for the failure can be displayed by using `clnt_perror()`, or an error string
can be returned through `clnt_sperror()`.

The second type of error is caused by the server itself. In Example 26, "`printmsg_1` for
Superuser," on page 56, an error can be returned by `opendir()`. The handling of these
errors is application specific and is the responsibility of the programmer.

Note that you will be unable to link the client and server programs to each other if you are using
the `-C` option, because of the `-_svc` suffix added to the server-side routines.

◆ ◆ ◆ **C H A P T E R 4**

4

Programmer's Interface to RPC

This chapter addresses the C interface to RPC and describes how to write network applications using RPC. For a complete specification of the routines in the RPC library, see the rpc(3NSL) and related man pages.

The topics covered in this chapter include:

- "Simplified Interface" on page 61
- "Standard Interfaces" on page 69
- "Testing Programs Using Low-Level Raw RPC" on page 84
- "MT User Mode" on page 141

Note - The client and server interfaces described in this chapter are multithread safe, except where noted (such as raw mode). This designation means that applications that contain RPC function calls can be used freely in a multithreaded application.

Simplified Interface

The simplified interface is the level to use if you do not require the use of any other RPC routines. This level also limits control of the underlying communications mechanisms. You can rapidly develop a program at this level, and the development is directly supported by the rpcgen compiler. For most applications, rpcgen and its facilities are sufficient.

Some RPC services are not available as C functions, but they are available as RPC programs. The simplified interface library routines provide direct access to the RPC facilities for programs that do not require fine levels of control. Routines such as rusers() are in the RPC services library librpcsvc. The following code example is a program that displays the number of users on a remote host. It calls the RPC library routine rusers().

EXAMPLE 28   rusers Program

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

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

```
#include <stdio.h>

/*
 * a program that calls the
 * rusers() service
 */

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

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

Compile the program in Example 28, "rusers Program," on page 61 by typing:

```
cc program.c -lrpcsvc -lnsl
```

Client Side of Simplified Interface

Just one function exists on the client side of the simplified interface: `rpc_call()`. It has nine parameters:

```
int 0 or error code
rpc_call (
      char  *host  /* Name of server host */
      rpcprog_t prognum  /* Server program number */
      rpcvers_t versnum  /* Server version number */
      rpcproc_t procnum  /* Server procedure number */
      xdrproc_t inproc  /* XDR filter to encode arg */
      char  *in  /* Pointer to argument */
      xdr_proc_t outproc  /* Filter to decode result */
      char  *out  /* Address to store result */
      char  *nettype /*For transport selection */
);
```

The `rpc_call()` function calls the procedure specified by *prognum*, *versum*, and *procnum* on the *host*. The argument to be passed to the remote procedure is pointed to by the *in* parameter, and *inproc* is the XDR filter to encode this argument. The *out* parameter is an address where

the result from the remote procedure is to be placed. *outproc* is an XDR filter that decodes the result and places it at this address.

The client blocks on `rpc_call()` until it receives a reply from the server. If the server accepts, it returns `RPC_SUCCESS` with the value of zero. The server returns a non-zero value if the call was unsuccessful. You can cast this value to the type `clnt_stat`, an enumerated type defined in the RPC include files and interpreted by the `clnt_sperrno()` function. This function returns a pointer to a standard RPC error message corresponding to the error code.

In the example, all "visible" transports listed in `/etc/netconfig` are tried. Adjusting the number of retries requires use of the lower levels of the RPC library.

Multiple arguments and results are handled by collecting them in structures.

The following code example changes the code in to use the simplified interface.

EXAMPLE 29   `rusers` Program Using Simplified Interface

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

/* A program that calls the RUSERSPROG RPC program */

main(argc, argv)
 int argc;
 char **argv;
{
 unsigned int nusers;
 enum clnt_stat cs;

 if (argc != 2) {
     fprintf(stderr, "usage: rusers hostname\n");
     exit(1);
 }
 if( cs = rpc_call(argv[1], RUSERSPROG,
                 RUSERSVERS, RUSERSPROC_NUM, xdr_void,
                 (char *)0, xdr_u_int, (char *)&nusers,
                 "visible")  !=  RPC_SUCCESS )  {
         clnt_perrno(cs);
         exit(1);
 }
 fprintf(stderr, "%d users on %s\n", nusers, argv[1] );
 exit(0);
}
```

Data types can be represented differently on different systems. Therefore, `rpc_call()` needs both the type of the RPC argument and a pointer to it. `rpc_call()` also needs this information

for the result. For `RUSERSPROC_NUM`, the return value is an `unsigned int`, so the first return parameter of `rpc_call()` is `xdr_u_int`, which is for an `unsigned int`, and the second return parameter is `&nusers`, which points to `unsigned int` storage. Because `RUSERSPROC_NUM` has no argument, the XDR encoding function of `rpc_call()` is `xdr_void()` and its argument is `NULL`.

Server Side of the Simplified Interface

The server program using the simplified interface is straightforward. The server calls `rpc_reg()` to register the procedure to be called. It then calls `svc_run()`, the RPC library's remote procedure dispatcher, to wait for requests to arrive.

`rpc_reg()` has the following arguments:

```
rpc_reg (
     rpcprog_t    prognum      /* Server program number */
     rpcvers_t    versnum      /* Server version number */
     rpcproc_t    procnum      /* server procedure number */
     char         *procname    /* Name of remote function */
     xdrproc_t    inproc       /* Filter to encode arg */
     xdrproc_t    outproc      /* Filter to decode result */
     char         *nettype     /* For transport selection */
);
```

`svc_run()` invokes service procedures in response to RPC call messages. The dispatcher in `rpc_reg()` decodes remote procedure arguments and encodes results, using the XDR filters specified when the remote procedure was registered. Some notes about the server program include:

- Most RPC applications follow the naming convention of appending a `_1` to the function name. The sequence `_n` is added to the procedure names to indicate the version number *n* of the service.
- The argument and result are passed as addresses. This is true for all functions that are called remotely. Passing `NULL` as a result of a function means that no reply is sent to the client, because `NULL` indicates that there is no reply to send.
- The result must exist in static data space because its value is accessed after the actual procedure has exited. The RPC library function that builds the RPC reply message accesses the result and sends the value back to the client.
- Only a single argument is allowed. If there are multiple elements of data, they should be wrapped inside a structure that can then be passed as a single entity.
- The procedure is registered for each transport of the specified type. If the type parameter is `(char *)NULL`, the procedure is registered for all transports specified in `NETPATH`.

Hand-Coded Registration Routine

You can sometimes implement faster or more compact code than can rpcgen. rpcgen handles the generic code-generation cases. The following program is an example of a hand-coded registration routine. It registers a single procedure and enters svc_run() to service requests.

```
#include <stdio.h>
 #include <rpc/rpc.h>
 #include <rpcsvc/rusers.h>
 void *rusers();

 main()
 {
  if(rpc_reg(RUSERSPROG, RUSERSVERS,
   RUSERSPROC_NUM, rusers,
    xdr_void, xdr_u_int,
   "visible") == -1) {
   fprintf(stderr, "Couldn't Register\n");
   exit(1);
  }
  svc_run();     /* Never returns */
  fprintf(stderr, "Error: svc_run
  returned!\n");
  exit(1);
}
```

rpc_reg() can be called as many times as is needed to register different programs, versions, and procedures.

Passing Arbitrary Data Types

Data types passed to and received from remote procedures can be any of a set of predefined types, or can be programmer-defined types. RPC handles arbitrary data structures, regardless of the byte orders or structure layout conventions of different systems. RPC always converts these structures to a standard transfer format called external data representation (XDR) before sending them over the transport. The conversion from a system representation to XDR is called serializing, and the reverse process is called deserializing.

The translator arguments of rpc_call() and rpc_reg() can specify an XDR primitive procedure, like xdr_u_int(), or a programmer-supplied routine that processes a complete argument structure. Argument processing routines must take only two arguments: a pointer to the result and a pointer to the XDR handle.

The XDR Primitive Type Routines are:

xdr_int()   xdr_double()

```
xdr_netobj()                        xdr_u_short()
xdr_u_long()                        xdr_wrapstring()
xdr_enum()                          xdr_char()
xdr_long()                          xdr_quadruple()
xdr_float()                         xdr_u_char()
xdr_u_int()                         xdr_void()
xdr_bool()                          xdr_hyper()
xdr_short()                         xdr_u_hyper()
```

The fixed-width integer types found in `int_types.h`, the routines `xdr_char()`, `xdr_short()`, `xdr_int()`, and `xdr_hyper()` (and the unsigned versions of each) have equivalent functions with names familiar to ANSI C, as indicated in the following table.

TABLE 12   Primitive Type Equivalences

Function	Equivalent
xdr_char()	xdr_int8_t()
xdr_u_char()	xdr_u_int8_t()
xdr_short()	xdr_int16_t()
xdr_u_short()	xdr_u_int16_t()
xdr_int()	xdr_int32_t()
xdr_u_int()	xdr_u_int32_t()
xdr_hyper()	xdr_int64_t()
xdr_u_hyper()	xdr_u_int64_t()

The nonprimitive `xdr_string()`, which takes more than two parameters, is called from `xdr_wrapstring()`.

The following example of a programmer-supplied routine contains the calling arguments of a procedure.

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

The XDR routine `xdr_simple()` translates the argument structure as shown in the following code example.

EXAMPLE 30   `xdr_simple` Routine

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

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

rpcgen can automatically generate an equivalent routine.

An XDR routine returns nonzero (a C TRUE) if it completes successfully, and zero otherwise. A complete description of XDR is provided in Appendix C, "XDR Protocol Specification".

The following list shows prefabricated routines:

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

For example, to send a variable-sized array of integers, the routine is packaged in a structure containing the array and its length:

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

Translate the array with xdr_varintarr(), as shown in the following code example.

EXAMPLE 31   xdr_varintarr Syntax Use

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

```
}
```

The arguments of `xdr_array()` are the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum array size, the size of each array element, and a pointer to the XDR routine to translate each array element. If the size of the array is known in advance, use `xdr_vector()`, as shown in the following code example.

EXAMPLE 32   `xdr_vector` Syntax Use

```
int intarr[SIZE];

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

XDR converts quantities to 4-byte multiples when serializing. For arrays of characters, each character occupies 32 bits. `xdr_bytes()` packs characters. It has four parameters similar to the first four parameters of `xdr_array()`.

Null-terminated strings are translated by `xdr_string()`, which is like `xdr_bytes()` with no length parameter. On serializing `xdr_string()` gets the string length from `strlen()`, and on deserializing it creates a null-terminated string.

The following example calls the built-in functions `xdr_string()` and `xdr_reference()`, which translates pointers to pass a string, and `struct simple` from previous examples.

EXAMPLE 33   `xdr_reference` Syntax Use

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

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

```
}
```

Note that `xdr_simple()` could have been called here instead of `xdr_reference()`.

Standard Interfaces

Interfaces to standard levels of the RPC package provide increasing control over RPC communications. Programs that use this control are more complex. Effective programming at these lower levels requires more knowledge of computer network fundamentals. The top, intermediate, expert, and bottom levels are part of the standard interfaces.

This section describes how to control RPC details by using lower levels of the RPC library. For example, you can select the transport protocol, which can be done at the simplified interface level only through the `NETPATH` variable. You should be familiar with the top-level interface (TLI) in order to use these routines.

The routines shown below cannot be used through the simplified interface because they require a transport handle. For example, there is no way to allocate and free memory while serializing or deserializing with XDR routines at the simplified interface.

```
clnt_call()
clnt_destroy()
clnt_control()
clnt_perrno()
clnt_pcreateerror()
clnt_perror()
svc_destroy()
```

Top-Level Interface

At the top level, the application can specify the type of transport to use but not the specific transport. This level differs from the simplified interface in that the application creates its own transport handles in both the client and server.

Client Side of the Top-Level Interface

Assume the header file in the following code example.

EXAMPLE 34   `time_prot.h` Header File

```
/* time_prot.h */
```

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

struct timev {
    int second;
    int minute;
    int hour;
}; typedef struct timev timev;
bool_t xdr_timev();

#define TIME_PROG 0x40000001
#define TIME_VERS 1
#define TIME_GET 1
```

The following example shows the client side of a trivial date service using top-level service routines. The transport type is specified as an invocation argument of the program.

EXAMPLE 35   Client for Trivial Date Service

```
#include <stdio.h>
#include "time_prot.h"

#define TOTAL (30)
/*
 * Caller of trivial date service
 * usage: calltime hostname
 */
main(argc, argv)
 int argc;
 char *argv[];
{
 struct timeval time_out;
 CLIENT *client;
 enum clnt_stat stat;
 struct timev timev;
 char *nettype;

 if (argc != 2 && argc != 3) {
  fprintf(stderr,"usage:%s host[nettype]\n" ,argv[0]);
  exit(1);
 }
 if (argc == 2)
  nettype = "netpath";  /* Default */
 else
  nettype = argv[2];
 client = clnt_create(argv[1], TIME_PROG, TIME_VERS, nettype);
 if (client == (CLIENT *) NULL) {
  clnt_pcreateerror("Couldn't create client");
  exit(1);
```

```
 }
 time_out.tv_sec = TOTAL;
 time_out.tv_usec = 0;
 stat = clnt_call( client, TIME_GET,
  xdr_void, (caddr_t)NULL,
  xdr_timev, (caddr_t)&timev,
  time_out);
 if (stat != RPC_SUCCESS) {
  clnt_perror(client, "Call failed");
  exit(1);
 }
 fprintf(stderr,"%s: %02d:%02d:%02d GMT\n",
  nettype timev.hour, timev.minute,
  timev.second);
 (void) clnt_destroy(client);
 exit(0);
}
```

If *nettype* is not specified in the invocation of the program, the string *netpath* is substituted. When RPC libraries routines encounter this string, the value of the NETPATH environment variable governs transport selection.

If the client handle cannot be created, display the reason for the failure with clnt_pcreateerror(). You can also get the error status by reading the contents of the global variable *rpc_createerr*.

After the client handle is created, clnt_call() is used to make the remote call. Its arguments are the remote procedure number, an XDR filter for the input argument, the argument pointer, an XDR filter for the result, the result pointer, and the time-out period of the call. The program has no arguments, so xdr_void() is specified. Clean up by calling clnt_destroy().

To bound the time allowed for client handle creation in the previous example to 30 seconds, replace the call to clnt_create() with a call to clnt_create_timed() as shown in the following code segment:

```
struct timeval timeout;
timeout.tv_sec = 30;  /* 30 seconds */
timeout.tv_usec = 0;

client = clnt_create_timed(argv[1], TIME_PROG, TIME_VERS, nettype,
                           &timeout);
```

The following example shows a top-level implementation of a server for the trivial date service.

EXAMPLE 36   Server for Trivial Date Service

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "time_prot.h"
```

```
static void time_prog();

main(argc,argv)
 int argc;
 char *argv[];
{
 int transpnum;
 char *nettype;

 if (argc > 2) {
  fprintf(stderr, "usage: %s [nettype]\n", argv[0]);
  exit(1);
 }
 if (argc == 2)
  nettype = argv[1];
 else
  nettype = "netpath";  /* Default */
 transpnum = svc_create(time_prog,TIME_PROG,TIME_VERS,nettype);
 if (transpnum == 0) {
  fprintf(stderr,"%s: cannot create %s service.\n",
     argv[0], nettype);
  exit(1);
 }
 svc_run();
}

/*
 * The server dispatch function
 */
static void
time_prog(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 struct timev rslt;
 time_t thetime;

 switch(rqstp->rq_proc) {
  case NULLPROC:
   svc_sendreply(transp, xdr_void, NULL);
   return;
  case TIME_GET:
   break;
  default:
   svcerr_noproc(transp);
   return;
  }
 thetime = time((time_t *) 0);
 rslt.second = thetime % 60;
 thetime /= 60;
 rslt.minute = thetime % 60;
```

```
 thetime /= 60;
 rslt.hour = thetime % 24;
 if (!svc_sendreply( transp, xdr_timev, &rslt)) {
  svcerr_systemerr(transp);
  }
}
```

`svc_create()` returns the number of transports on which it created server handles. `time_prog()` is the service function called by `svc_run()` when a request specifies its program and version numbers. The server returns the results to the client through `svc_sendreply()`.

When you use `rpcgen` to generate the dispatch function, `svc_sendreply()` is called after the procedure returns. Therefore, *rslt* in this example must be declared `static` in the actual procedure. `svc_sendreply()` is called from inside the dispatch function, so `rslt` is not declared `static`.

In this example, the remote procedure takes no arguments. When arguments must be passed, the calls listed below fetch, deserialize (XDR decode), and free the arguments.

```
svc_getargs( SVCXPRT_handle, XDR_filter, argument_pointer);
svc_freeargs( SVCXPRT_handle, XDR_filter argument_pointer );
```

Intermediate-Level Interface

At the intermediate level, the application directly chooses the transport to use.

Client Side of the Intermediate-Level Interface

The following example shows the client side of the time service from "Top-Level Interface" on page 69, written at the intermediate level of RPC. In this example, the user must name the transport over which the call is made on the command line.

EXAMPLE 37   Client for Time Service, Intermediate Level

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>  /* For netconfig structure */
#include "time_prot.h"

#define TOTAL (30)

main(argc,argv)
 int argc;
 char *argv[];
{
 CLIENT *client;
```

```
struct netconfig *nconf;
char *netid;
/* Declarations from previous example */

if (argc != 3) {
 fprintf(stderr, "usage: %s host netid\n", argv[0]);
 exit(1);
}
netid = argv[2];
if ((nconf = getnetconfigent( netid)) ==
    (struct netconfig *) NULL) {
 fprintf(stderr, "Bad netid type: %s\n", netid);
 exit(1);
}
client = clnt_tp_create(argv[1], TIME_PROG,
         TIME_VERS, nconf);
if (client == (CLIENT *) NULL) {
 clnt_pcreateerror("Could not create client");
 exit(1);
}
freenetconfigent(nconf);

 /* Same as previous example after this point */
}
```

In this example, the netconfig structure is obtained by a call to getnetconfigent(netid). See
the getnetconfig(3NSL) man page and *Oracle Solaris 11.3 Programming Interfaces Guide* for
more details. At this level, the program explicitly selects the network.

To bound the time allowed for client handle creation in the previous example to 30 seconds,
replace the call to clnt_tp_create() with a call to clnt_tp_create_timed() as shown in the
following code segment:

```
 struct timeval timeout;
 timeout.tv_sec = 30; /* 30 seconds */
 timeout.tv_usec = 0;

 client = clnt_tp_create_timed(argv[1],
    TIME_PROG, TIME_VERS, nconf,
    &timeout);
```

Server Side of the Intermediate-Level Interface

The following example shows the corresponding server. The command line that starts the
service must specify the transport over which the service is provided.

EXAMPLE 38   Server for Time Service, Intermediate Level

```
/*
```

```
 * This program supplies Greenwich mean
 * time to the client that invokes it.
 * The call format is: server netid
 */
#include <stdio.h>
#include <rpc/rpc.h>

#include <netconfig.h>    /* For netconfig structure */
#include "time_prot.h"

static void time_prog();

main(argc, argv)
 int argc;
 char *argv[];
{
 SVCXPRT *transp;
 struct netconfig *nconf;

 if (argc != 2) {
  fprintf(stderr, "usage: %s netid\n", argv[0]);
  exit(1);
 }
 if ((nconf = getnetconfigent( argv[1])) ==
                                   (struct netconfig *) NULL) {
  fprintf(stderr, "Could not find info on %s\n", argv[1]);
  exit(1);
 }
 transp = svc_tp_create(time_prog, TIME_PROG,
         TIME_VERS, nconf);
 if (transp == (SVCXPRT *) NULL) {
  fprintf(stderr, "%s: cannot create %s service\n",
                        argv[0], argv[1]);
  exit(1)
 }
 freenetconfigent(nconf);
 svc_run();
}

 static
 void time_prog(rqstp, transp)
  struct svc_req *rqstp;
  SVCXPRT *transp;
{
/* Code identical to Top Level version */
```

Expert-Level Interface

At the expert level, network selection is done the same as at the intermediate level. The only difference is in the increased level of control that the application has over the details of the CLIENT and SVCXPRT handles. These examples illustrate this control, which is exercised using the clnt_tli_create() and svc_tli_create() routines. For more information on TLI, see *Oracle Solaris 11.3 Programming Interfaces Guide*.

Client Side of the Expert-Level Interface

Example 39, "Client for RPC Lower Level," on page 76 shows a version of clntudp_create(), the client creation routine for UDP transport, using clnt_tli_create(). The example shows how to do network selection based on the family of the transport you choose. clnt_tli_create() is used to create a client handle and to:

- Pass an open TLI file descriptor, which might or might not be bound
- Pass the server's address to the client
- Specify the send and receive buffer size

EXAMPLE 39   Client for RPC Lower Level

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>
/*
 * In earlier implementations of RPC,
 * only TCP/IP and UDP/IP were supported.
 * This version of clntudp_create()
 * is based on TLI/Streams.
 */
CLIENT *
clntudp_create(raddr, prog, vers, wait, sockp)
 struct sockaddr_in *raddr;  /* Remote address */
 rpcprog_t prog;     /* Program number */
 prcvers_t vers;     /* Version number */
 struct timeval wait;   /* Time to wait */
 int *sockp;         /* fd pointer */
{
 CLIENT *cl;         /* Client handle */
 int madefd = FALSE;   /* Is fd opened here */
 int fd = *sockp;   /* TLI fd */
 struct t_bind *tbind;   /* bind address */
 struct netconfig *nconf;  /* netconfig structure */
 void *handlep;

 if ((handlep = setnetconfig() ) == (void *) NULL) {
```

```
 /* Error starting network configuration */
 rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
 return((CLIENT *) NULL);
}
/*
 * Try all the transports until it gets one that is
 * connectionless, family is INET, and preferred name is UDP
 */
while (nconf = getnetconfig( handlep)) {
 if ((nconf->nc_semantics == NC_TPI_CLTS) &&

      (strcmp( nconf->nc_protofmly, NC_INET ) == 0) &&
      (strcmp( nconf->nc_proto, NC_UDP ) == 0))
  break;
}
if (nconf == (struct netconfig *) NULL)
 rpc_createerr.cf_stat = RPC_UNKNOWNPROTO;
 goto err;
}
if (fd == RPC_ANYFD) {
 fd = t_open(nconf->nc_device, O_RDWR, &tinfo);
 if (fd == -1) {
  rpc_createerr.cf_stat = RPC_SYSTEMERROR;
  goto err;
 }
}
if (raddr->sin_port == 0) { /* remote addr unknown */
 u_short sport;
 /*
  * rpcb_getport() is a user-provided routine that calls
  * rpcb_getaddr and translates the netbuf address to port
  * number in host byte order.
  */
 sport = rpcb_getport(raddr, prog, vers, nconf);
 if (sport == 0) {
  rpc_createerr.cf_stat = RPC_PROGUNAVAIL;
  goto err;
 }
 raddr->sin_port = htons(sport);
}
/* Transform sockaddr_in to netbuf */
tbind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
if (tbind == (struct t_bind *) NULL)
 rpc_createerr.cf_stat = RPC_SYSTEMERROR;
 goto err;
}
if (t_bind->addr.maxlen < sizeof( struct sockaddr_in))
 goto err;
(void) memcpy( tbind->addr.buf, (char *)raddr,
                sizeof(struct sockaddr_in));
tbind->addr.len = sizeof(struct sockaddr_in);
```

```
/* Bind fd */
if (t_bind( fd, NULL, NULL) == -1) {
 rpc_createerr.ct_stat = RPC_TLIERROR;
 goto err;
}
cl = clnt_tli_create(fd, nconf, &(tbind->addr), prog, vers,
                        tinfo.tsdu, tinfo.tsdu);
/* Close the netconfig file */
(void) endnetconfig( handlep);
(void) t_free((char *) tbind, T_BIND);
if (cl) {
 *sockp = fd;
 if (madefd == TRUE) {
  /* fd should be closed while destroying the handle */
  (void)clnt_control(cl,CLSET_FD_CLOSE, (char *)NULL);
 }
 /* Set the retry time */
 (void) clnt_control( l, CLSET_RETRY_TIMEOUT,
                        (char *) &wait);
 return(cl);
}
err:
 if (madefd == TRUE)
  (void) t_close(fd);
 (void) endnetconfig(handlep);
 return((CLIENT *) NULL);
}
```

The network is selected using `setnetconfig()`, `getnetconfig()`, and `endnetconfig()`.
`endnetconfig()` is not called until after the call to `clnt_tli_create()`, near the end of the
example.

`clntudp_create()` can be passed an open TLI *fd*. If passed none (`fd == RPC_ANYFD`),
`clntudp_create()` opens its own using the `netconfig` structure for UDP to find the name of
the device to pass to `t_open()`.

If the remote address is not known (`raddr->sin_port == 0`), it is obtained from the remote
`rpcbind`.

After the client handle has been created, you can modify it using calls to `clnt_control()`.
The RPC library closes the file descriptor when destroying the handle, as it does with a call to
`clnt_destroy()` when it opens the *fd* itself. The RPC library then sets the retry timeout period.

Server Side of the Expert-Level Interface

shows the server side of
. It is called `svcudp_create()`. The server
side uses `svc_tli_create()`.

svc_tli_create() is used when the application needs a fine degree of control, particularly to:

- Pass an open file descriptor to the application.
- Pass the user's bind address.
- Set the send and receive buffer sizes. The *fd* argument can be unbound when passed in. It is then bound to a given address, and the address is stored in a handle. If the bind address is set to NULL and the *fd* is initially unbound, it is bound to any suitable address.

Use rpcb_set() to register the service with rpcbind.

EXAMPLE 40   Server for RPC Lower Level

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <netconfig.h>
#include <netinet/in.h>

SVCXPRT *
svcudp_create(fd)
 register int fd;
{
 struct netconfig *nconf;
 SVCXPRT *svc;
 int madefd = FALSE;
 int port;
 void *handlep;
 struct  t_info tinfo;

 /* If no transports available */
 if ((handlep = setnetconfig() ) == (void *) NULL) {
  nc_perror("server");
  return((SVCXPRT *) NULL);
 }
 /*
  * Try all the transports until it gets one which is
  * connectionless, family is INET and, name is UDP
  */
 while (nconf = getnetconfig( handlep)) {
  if ((nconf->nc_semantics == NC_TPI_CLTS) &&
      (strcmp( nconf->nc_protofmly, NC_INET) == 0 )&&
      (strcmp( nconf->nc_proto, NC_UDP) == 0 ))
   break;
 }
 if (nconf == (struct netconfig *) NULL) {
  endnetconfig(handlep);
  return((SVCXPRT *) NULL);
 }
 if (fd == RPC_ANYFD) {
  fd = t_open(nconf->nc_device, O_RDWR, &tinfo);
  if (fd == -1) {
```

```
   (void) endnetconfig();
   return((SVCXPRT *) NULL);
 }
 madefd = TRUE;
} else
 t_getinfo(fd, &tinfo);
svc = svc_tli_create(fd, nconf, (struct t_bind *) NULL,
                       tinfo.tsdu, tinfo.tsdu);
(void) endnetconfig(handlep);
if (svc == (SVCXPRT *) NULL) {
 if (madefd)
   (void) t_close(fd);
 return((SVCXPRT *)NULL);
 }
 return (svc);
}
```

The network selection here is accomplished similar to `clntudp_create()`. The file descriptor is not bound explicitly to a transport address because `svc_tli_create()` does that.

`svcudp_create()` can use an open *fd*. It opens one itself using the selected `netconfig` structure if none is provided.

Bottom-Level Interface

The bottom-level interface to RPC enables the application to control all options. `clnt_tli_create()` and the other expert-level RPC interface routines are based on these routines. You rarely use these low-level routines.

Bottom-level routines create internal data structures, buffer management, RPC headers, and so on. Callers of these routines, like the expert-level routine `clnt_tli_create()`, must initialize the *cl_netid* and *cl_tp* fields in the client handle. For a created handle, *cl_netid* is the network identifier (for example, `udp`) of the transport and `cl_tp` is the device name of that transport (for example, /dev/udp). The routines `clnt_dg_create()` and `clnt_vc_create()` set the *clnt_ops* and *cl_private* fields.

Client Side of the Bottom-Level Interface

The following code example shows calls to `clnt_vc_create()` and `clnt_dg_create()`.

EXAMPLE 41   Client for Bottom Level

```
/*
 * variables are:
 * cl: CLIENT *
 * tinfo: struct t_info returned from either t_open or t_getinfo
```

```
 * svcaddr: struct netbuf *
 */
 switch(tinfo.servtype) {
  case T_COTS:
  case T_COTS_ORD:
   cl = clnt_vc_create(fd, svcaddr,
    prog, vers, sendsz, recvsz);
   break;
  case T_CLTS:
   cl = clnt_dg_create(fd, svcaddr,
    prog, vers, sendsz, recvsz);
   break;
  default:
   goto err;
 }
```

These routines require that the file descriptor be open and bound. *svcaddr* is the address of the server.

Server Side of the Bottom-Level Interface

The following code example is an example of creating a bottom-level server.

EXAMPLE 42   Server for Bottom Level

```
/*
 * variables are:
 * xprt: SVCXPRT *
 */
switch(tinfo.servtype) {
 case T_COTS_ORD:
 case T_COTS:
  xprt = svc_vc_create(fd, sendsz, recvsz);
  break;
 case T_CLTS:
  xprt = svc_dg_create(fd, sendsz, recvsz);
  break;
 default:
  goto err;
}
```

Server Caching

svc_dg_enablecache() initiates service caching for datagram transports. Caching should be used only in cases where a server procedure is a "once only" kind of operation. Executing a cached server procedure multiple times yields different results.

```
svc_dg_enablecache(xprt, cache_size)
    SVCXPRT *xprt;
    unsigned int cache_size;
```

This function allocates a duplicate request cache for the service endpoint *xprt,* large enough to hold *cache_size* entries. A duplicate request cache is needed if the service contains procedures with varying results. After caching is enabled, it cannot be disabled.

Low-Level Data Structures

The following data structure information is for reference only. The implementation might change.

The first structure is the client RPC handle, defined in `<rpc/clnt.h>`. Low-level implementations must provide and initialize one handle per connection, as shown in the following code example.

EXAMPLE   43   RPC Client Handle Structure

```
typedef struct {
 AUTH *cl_auth;        /* authenticator */
 struct clnt_ops {
  enum clnt_stat (*cl_call)();   /* call remote procedure */
  void  (*cl_abort)();   /* abort a call */
  void  (*cl_geterr)();   /* get specific error code */
  bool_t  (*cl_freeres)();  /* frees results */
  void  (*cl_destroy)();  /* destroy this structure */
  bool_t  (*cl_control)();  /* the ioctl() of rpc */
 } *cl_ops;
 caddrt_t  cl_private;   /* private stuff */
 char   *cl_netid;   /* network token */
 char   *cl_tp;    /* device name */
} CLIENT;
```

The first field of the client-side handle is an authentication structure, defined in `<rpc/auth. h>`. By default, this field is set to `AUTH_NONE`. A client program must initialize `cl_auth` appropriately, as shown in the following code example.

EXAMPLE   44   Client Authentication Handle

```
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 varifier */
  int (*ah_refresh)();   /* refresh credentials */
  void (*ah_destroy)();   /* destroy this structure */
 } *ah_ops;
 caddr_t ah_private;
} AUTH;
```

In the AUTH structure, ah_cred contains the caller's credentials, and ah_verf contains the data to
verify the credentials. See "Authentication" on page 100 for details.

The following code example shows the server transport handle.

EXAMPLE 45   Server Transport Handle

```
typedef struct {
 int  xp_fd;
#define xp_sock  xp_fd
 u_short xp_port;  /* associated port number. Obsoleted */
 struct xp_ops {
     bool_t (*xp_recv)();  /* receive incoming requests */
     enum xprt_stat (*xp_stat)(); /* get transport status */
     bool_t (*xp_getargs)(); /* get arguments */
     bool_t (*xp_reply)();  /* send reply */
     bool_t (*xp_freeargs)(); /* free mem alloc for args */
     void (*xp_destroy)(); /* destroy this struct */
 } *xp_ops;
 int  xp_addrlen;  /* length of remote addr. Obsolete */
 char  *xp_tp;   /* transport provider device name */
 char  *xp_netid;  /* network token */
 struct netbuf  xp_ltaddr;  /* local transport address */
 struct netbuf  xp_rtaddr;  /* remote transport address */
 char  xp_raddr[16]; /* remote address. Now obsoleted */
 struct opaque_auth xp_verf; /* raw response verifier */
 caddr_t  xp_p1;  /* private: for use by svc ops */
 caddr_t  xp_p2;  /* private: for use by svc ops */
 caddr_t  xp_p3;  /* private: for use by svc lib */
} SVCXPRT;
```

The following table shows the fields for the server transport handle.

xp_fd	The file descriptor associated with the handle. Two or more server handles can share the same file descriptor.
xp_netid	The network identifier (for example, udp) of the transport on which the handle is created and *xp_tp* is the device name associated with that transport.
xp_ltaddr	The server's own bind address.
xp_rtaddr	The address of the remote caller (and so can change from call to call).

`xp_netid xp_tp xp_ltaddr`	Initialized by `svc_tli_create()` and other expert-level routines.

The rest of the fields are initialized by the bottom-level server routines `svc_dg_create()` and `svc_vc_create()`.

For connection-oriented endpoints, the following fields are not valid until a connection has been requested and accepted for the server:

```
xp_fd
xp_ops()
xp_p1()
xp_p2
xp_verf()
xp_tp()
xp_ltaddr
xp_rtaddr()
xp_netid()
```

Testing Programs Using Low-Level Raw RPC

Two pseudo-RPC interface routines bypass all the network software. The routines shown in `clnt_raw_create()` and `svc_raw_create()` do not use any real transport.

Note - Do not use raw mode on production systems. Raw mode is intended as a debugging aid only. Raw mode is not MT safe.

The following code example is compiled and linked using the following makefile:

```
all: raw
CFLAGS += -g
raw: raw.o
cc -g -o raw raw.o -lnsl
```

EXAMPLE 46   Simple Program Using Raw RPC

```
/*
 * A simple program to increment a number by 1
 */

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

#include <rpc/raw.h>
#define prognum 0x40000001
```

```
#define versnum 1
#define INCR 1

struct timeval TIMEOUT = {0, 0};
static void server();

main (argc, argv)
 int argc;
 char **argv;
{
 CLIENT *cl;
 SVCXPRT *svc;
 int num = 0, ans;
 int flag;

 if (argc == 2)
  num = atoi(argv[1]);
  svc = svc_raw_create();
 if (svc == (SVCXPRT *) NULL) {
  fprintf(stderr, "Could not create server handle\n");
  exit(1);
 }
 flag = svc_reg( svc, prognum, versnum, server,
        (struct netconfig *) NULL );
   if (flag == 0) {
     fprintf(stderr, "Error: svc_reg failed.\n");
     exit(1);
 }
 cl = clnt_raw_create( prognum, versnum );
 if (cl == (CLIENT *) NULL) {
  clnt_pcreateerror("Error: clnt_raw_create");
  exit(1);
 }
 if (clnt_call(cl, INCR, xdr_int, (caddr_t) &num, xdr_int,

      (caddr_t) &ans, TIMEOUT)
   != RPC_SUCCESS) {
  clnt_perror(cl, "Error: client_call with raw");
  exit(1);
 }
 printf("Client: number returned %d\n", ans);
 exit(0);
}

static void
server(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 int num;
```

```
    fprintf(stderr, "Entering server procedure.\n");

switch(rqstp->rq_proc) {
 case NULLPROC:
  if (svc_sendreply( transp, xdr_void,
   (caddr_t) NULL) == FALSE) {
   fprintf(stderr, "error in null proc\n");
   exit(1);
  }
  return;
 case INCR:
  break;
 default:
  svcerr_noproc(transp);
  return;
}
if (!svc_getargs( transp, xdr_int, &num)) {
 svcerr_decode(transp);
 return;
}
   fprintf(stderr, "Server procedure: about to increment.\n");
num++;
if (svc_sendreply(transp, xdr_int, &num) == FALSE) {
 fprintf(stderr, "error in sending answer\n");
 exit (1);
}
   fprintf(stderr, "Leaving server procedure.\n");
}
```

Note the following points about the example:

- The server must be created before the client.

- `svc_raw_create()` has no parameters.

- The server is not registered with `rpcbind`. The last parameter to `svc_reg()` is (`struct netconfig *`) `NULL`.

- `svc_run()` is not called.

- All the RPC calls occur within the same thread of control.

- The server-dispatch routine is the same as for normal RPC servers.

Connection-Oriented Transports

copies a file from one host to another. The RPC send() call reads standard input and sends the data to the server

receive(), which writes the data to standard output. This example also illustrates an XDR procedure that behaves differently on serialization and on deserialization. A connection-oriented transport is used.

EXAMPLE 47   Remote Copy (Two-Way XDR Routine)

```
/*
 * The xdr routine:
 *      on decode, read wire, write to fp
 *      on encode, read fp, write to wire
 */
#include <stdio.h>
#include <rpc/rpc.h>

bool_t
xdr_rcp(xdrs, fp)
 XDR *xdrs;
 FILE *fp;
{
 unsigned long size;
 char buf[BUFSIZ], *p;

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

In Example 48, "Remote Copy Client Routines," on page 88 and Example 49, "Remote Copy Server Routines," on page 89 , the serializing and deserializing are done only by the

xdr_rcp() routine shown in .

EXAMPLE 48   Remote Copy Client Routines

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

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

#include <sys/socket.h>
#include <sys/time.h>
#include "rcp.h"

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

 if (argc != 2 7) {
  fprintf(stderr, "usage: %s servername\n", argv[0]);
  exit(1);
 }
 if( callcots( argv[1], RCPPROG, RCPPROC, RCPVERS, xdr_rcp,
stdin,
    xdr_void, 0 ) != 0 )
  exit(1);
 exit(0);
}

callcots(host, prognum, procnum, versnum, inproc, in, outproc,
out)
 char *host, *in, *out;
 xdrproc_t inproc, outproc;
{
 enum clnt_stat clnt_stat;
 register CLIENT *client;
 struct timeval total_timeout;

 if ((client = clnt_create( host, prognum, versnum,
"circuit_v")
    == (CLIENT *) NULL)) {
  clnt_pcreateerror("clnt_create");
  return(-1);
 }
 total_timeout.tv_sec = 20;
 total_timeout.tv_usec = 0;
 clnt_stat = clnt_call(client, procnum, inproc, in, outproc,
```

```
out,
                            total_timeout);
 clnt_destroy(client);
 if (clnt_stat != RPC_SUCCESS)
  clnt_perror("callcots");
 return((int)clnt_stat);
}
```

The following code example defines the receiving routines. Note that in the server, `xdr_rcp()` did all the work automatically.

EXAMPLE 49   Remote Copy Server Routines

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

main()
{
 void  rcp_service();
 if (svc_create(rpc_service,RCPPROG,RCPVERS,"circuit_v") == 0) {
  fprintf(stderr, "svc_create: errpr\n");
  exit(1);
 }
 svc_run();                       /* never returns */
 fprintf(stderr, "svc_run should never return\n");
}

void
rcp_service(rqstp, transp)
 register struct svc_req *rqstp;
 register SVCXPRT *transp;
{
 switch(rqstp->rq_proc) {
  case NULLPROC:
   if (svc_sendreply(transp, xdr_void, (caddr_t) NULL)
                                   == FALSE)
    fprintf(stderr, "err: rcp_service");
   return;
  case RCPPROC:
   if (!svc_getargs( transp, xdr_rcp, stdout)) {
    svcerr_decode(transp);
    return();
   }
   if(!svc_sendreply(transp, xdr_void, (caddr_t) NULL)) {
    fprintf(stderr, "can't reply\n");
    return();
```

```
   }
   return();
  default:
   svcerr_noproc(transp);
   return();
 }
}
```

Memory Allocation With XDR

XDR routines normally serialize and deserialize data. XDR routines often automatically allocate memory and free automatically allocated memory. The convention is to use a `NULL` pointer to an array or structure to indicate that an XDR function must allocate memory when deserializing. The next example, `xdr_chararr1()`, processes a fixed array of bytes with length `SIZE` and cannot allocate memory if needed:

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

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

If space has already been allocated in *chararr*, it can be called from a server as follows.

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

Any structure through which data is passed to XDR or RPC routines must be allocated so that its base address is at an architecture-dependent boundary. An XDR routine that does the allocation must be written so that it can:

- Allocate memory when a caller requests
- Return the pointer to any memory it allocates

In the following example, the second argument is a `NULL` pointer, meaning that memory should be allocated to hold the data being deserialized.

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

```
   int len;

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

The corresponding RPC call is:

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

After use, free the character array through `svc_freeargs()`. `svc_freeargs()` does nothing if passed a `NULL` pointer as its second argument.

To summarize:

- An XDR routine normally serializes, deserializes, and frees memory.
- `svc_getargs()` calls the XDR routine to deserialize.
- `svc_freeargs()` calls the XDR routine to free memory.

5

Advanced RPC Programming Techniques

This section addresses areas of occasional interest to developers using the lower-level interfaces of the RPC package. The topics are:

- `poll()` on the server - How a server can call the dispatcher directly if calling `svc_run()` is not feasible
- Broadcast RPC - How to use the broadcast mechanisms
- Batching - How to improve performance by batching a series of calls
- Authentication - What authentication methods are available in this release
- Port monitors - How to interface with the `inetd` and `listener` port monitors
- Multiple program versions - How to service multiple program versions

`poll()` on the Server Side

This section applies only to servers running RPC in single-threaded (default) mode.

A process that services RPC requests and performs some other activity cannot always call `svc_run()`. If the other activity periodically updates a data structure, the process can set a `SIGALRM` signal before calling `svc_run()`. This process enables the signal handler to process the data structure and return control to `svc_run()` when done.

A process can bypass `svc_run()` and access the dispatcher directly with the `svc_getreqset()` call. The process must be given the file descriptors of the transport endpoints associated with the programs being waited on. Then the process can have its own `poll()` that waits on both the RPC file descriptors and its own descriptors.

Example 50, "`svc_run()` and `poll()`," on page 94 shows `svc_run()`. `svc_pollset` is an array of `pollfd` structures that is derived, through a call to `__rpc_select_to_poll()`, from `svc_fdset()`. The array can change every time any RPC library routine is called because descriptors are constantly being opened and closed. `svc_getreq_poll()` is called when `poll()` determines that an RPC request has arrived on some RPC file descriptors.

> **Note -** The `__rpc_dtbsize()` and `__rpc_select_to_poll()` functions are not part of the SVID, but they are available in the `libnsl` library. The descriptions of these functions are included here so that you can create versions of these functions for non-Oracle Solaris implementations.

Given an *fd_set* pointer and the number of bits to check in it, the `__rpc_select_to_poll` function initializes the supplied *pollfd* array for RPC's use. RPC polls only for input events. The number of *pollfd* slots that were initialized is returned. The arguments for this function are:

```
int __rpc_select_to_poll(int fdmax, fd_set *fdset,
    struct pollfd *pollset)
```

The `__rpc_dtbsize()` function calls the `getrlimit()` function to determine the maximum value that the system can assign to a newly created file descriptor. The result is cached for efficiency.

For more information on the SVID routines in this section, see the [rpc_svc_calls(3NSL)](#) and [poll(2)](#) man pages.

EXAMPLE 50   `svc_run()` and `poll()`

```
void
svc_run()
{
 int nfds;
 int dtbsize = __rpc_dtbsize();
 int i;
 struct pollfd svc_pollset[fd_setsize];

 for (;;) {
  /*
   * Check whether there is any server fd on which we may have
   * to wait.
   */
  nfds = __rpc_select_to_poll(dtbsize, &svc_fdset,
                                svc_pollset);
  if (nfds == 0)
   break; /* None waiting, hence quit */

  switch (i = poll(svc_pollset, nfds, -1)) {
  case -1:
   /*
    * We ignore all errors, continuing with the assumption
    * that it was set by the signal handlers (or any
    * other outside event) and not caused by poll().
    */
  case 0:
   continue;
  default:
   svc_getreq_poll(svc_pollset, i);
```

```
 }
 }
}
```

Broadcast RPC

When an RPC broadcast is issued, a message is sent to all rpcbind daemons on a network. An rpcbind daemon with which the requested service is registered forwards the request to the server. The main differences between broadcast RPC and normal RPC calls are:

- Normal RPC expects one answer; broadcast RPC expects many answers, one or more answer from each responding system.
- Broadcast RPC works only on connectionless protocols that support broadcasting, such as UDP.
- With broadcast RPC, all unsuccessful responses are filtered out. If a version mismatch occurs between the broadcaster and a remote service, the broadcaster is never contacted by the service.
- Only datagram services registered with rpcbind are accessible through broadcast RPC. Service addresses can vary from one host to another, so rpc_broadcast() sends messages to rpcbind's network address.
- The size of broadcast requests is limited by the maximum transfer unit (MTU) of the local network. The MTU for Ethernet is 1500 bytes.

The following code example shows how rpc_broadcast() is used and describes its arguments.

EXAMPLE 51   RPC Broadcast

```
/*
 * bcast.c: example of RPC broadcasting use.
 */

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

main(argc, argv)
 int argc;
 char *argv[];
{
 enum clnt_stat rpc_stat;
 rpcprog_t prognum;
 rpcvers_t vers;
 struct rpcent *re;

 if(argc != 3) {
  fprintf(stderr, "usage : %s RPC_PROG VERSION\n", argv[0]);
```

```
 exit(1);
}
if (isdigit( *argv[1]))
 prognum = atoi(argv[1]);
else {
 re = getrpcbyname(argv[1]);
 if (! re) {
  fprintf(stderr, "Unknown RPC service %s\n", argv[1]);
  exit(1);
 }
 prognum = re->r_number;
}
vers = atoi(argv[2]);
rpc_stat = rpc_broadcast(prognum, vers, NULLPROC, xdr_void,
           (char *)NULL, xdr_void, (char *)NULL, bcast_proc,
NULL);
if ((rpc_stat != RPC_SUCCESS) && (rpc_stat != RPC_TIMEDOUT)) {
 fprintf(stderr, "broadcast failed: %s\n",
          clnt_sperrno(rpc_stat));
 exit(1);
}
exit(0);
}
```

The function in Example 52, "Collect Broadcast Replies," on page 96 collects the replies to the broadcast. The normal operation is to collect either the first reply or all replies. `bcast_proc()` displays the IP address of the server that has responded. Because the function returns `FALSE` it continues to collect responses. The RPC client code continues to resend the broadcast until it times out.

EXAMPLE 52   Collect Broadcast Replies

```
bool_t
bcast_proc(res, t_addr, nconf)
    void *res;                 /* Nothing comes back */
    struct t_bind *t_addr;     /* Who sent us the reply */
    struct netconfig *nconf;
{
    register struct hostent *hp;
    char *naddr;

    naddr = taddr2naddr(nconf, &taddr->addr);
    if (naddr == (char *) NULL) {
        fprintf(stderr,"Responded: unknown\n");
    } else {
        fprintf(stderr,"Responded: %s\n", naddr);
        free(naddr);
    }
    return(FALSE);
}
```

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

Batching

RPC is designed so that clients send a call message and wait for servers to reply to the call. This procedure implies that a client is blocked while the server processes the call. This result is inefficient when the client does not need each message acknowledged.

RPC batching lets clients process asynchronously. RPC messages can be placed in a pipeline of calls to a server. Batching requires that:

- The server does not respond to any intermediate message.
- The pipeline of calls is transported on a reliable transport, such as TCP.
- The result's XDR routine in the calls is `NULL`.
- The RPC call's timeout is zero.

Because the server does not respond to each call, the client can send new calls in parallel with the server processing previous calls. The transport can buffer many call messages and send them to the server in one `write()` system call. This buffering decreases interprocess communication overhead and the total time of a series of calls. The client should end with a nonbatched call to flush the pipeline.

The following code example shows the unbatched version of the client. It scans the character array, *buf*, for delimited strings and sends each string to the server.

EXAMPLE 53   Unbatched Client

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

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

 if ((client = clnt_create( argv[1], WINDOWPROG, WINDOWVERS,
     "circuit_v")) == (CLIENT *) NULL) {
  clnt_pcreateerror("clnt_create");
```

```
 exit(1);
}

total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
while (scanf( "%s", s ) != EOF) {
 if (clnt_call(client, RENDERSTRING, xdr_wrapstring, &s,
    xdr_void, (caddr_t) NULL, total_timeout) != RPC_SUCCESS) {
  clnt_perror(client, "rpc");
  exit(1);
 }
}

clnt_destroy( client );
exit(0);
}
```

The following code example shows the batched version of the client. It does not wait after each
string is sent to the server. It waits only for an ending response from the server.

EXAMPLE 54   Batched Client

```
#include <stdio.h>

#include <rpc/rpc.h>
#include "windows.h"

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

 if ((client = clnt_create( argv[1], WINDOWPROG, WINDOWVERS,
    "circuit_v")) == (CLIENT *) NULL) {
  clnt_pcreateerror("clnt_create");
  exit(1);
 }
 timerclear(&total_timeout);
 while (scanf("%s", s) != EOF)
  clnt_call(client, RENDERSTRING_BATCHED, xdr_wrapstring,
            &s, xdr_void, (caddr_t) NULL, total_timeout);
 /* Now flush the pipeline */
 total_timeout.tv_sec = 20;
 clnt_stat = clnt_call(client, NULLPROC, xdr_void,
        (caddr_t) NULL, xdr_void, (caddr_t) NULL, total_timeout);
 if (clnt_stat != RPC_SUCCESS) {
```

```
  clnt_perror(client, "rpc");
  exit(1);
 }
 clnt_destroy(client);
 exit(0);
}
```

The following code example shows the dispatch portion of the batched server. Because the server sends no message, the clients are not notified of failures.

EXAMPLE 55   Batched Server

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

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, NULL))
    fprintf(stderr, "can't reply to RPC call\n");
   return;
  case RENDERSTRING:
   if (!svc_getargs( transp, xdr_wrapstring, &s)) {
    fprintf(stderr, "can't decode arguments\n");
    /* Tell caller an error occurred */
    svcerr_decode(transp);
    break;
   }
   /* Code here to render the string s */
   if (!svc_sendreply( transp, xdr_void, (caddr_t) NULL))
    fprintf( stderr, "can't reply to RPC call\n");
   break;
  case RENDERSTRING_BATCHED:
   if (!svc_getargs(transp, xdr_wrapstring, &s)) {
    fprintf(stderr, "can't decode arguments\n");
    /* Be silent in the face of protocol errors */
    break;
   }
   /* Code here to render string s, but send no reply! */
   break;
  default:
   svcerr_noproc(transp);
   return;
```

```
 }
 /* Now free string allocated while decoding arguments */
 svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Note - To illustrate the benefits of batching, Example 53, "Unbatched Client," on page 97 and Example 55, "Batched Server," on page 99 were completed to render the lines in a 25,144-line file. The rendering service throws away the lines. The batched version of the application is four times as fast as the unbatched version.

Authentication

Just as you can use different transports when creating RPC clients and servers, you can associate different "flavors" of authentication with RPC clients. The authentication subsystem of RPC is open ended. So, RPC can support many flavors of authentication. Appendix B, "RPC Protocol and Language Specification" further defines the authentication protocols.

Oracle RPC currently supports the authentication flavors shown in the following table.

TABLE 13   Authentication Methods Supported by Oracle RPC

Method	Description
AUTH_NONE	Default. No authentication performed.
AUTH_SYS	An authentication flavor based on the process permissions authentication in the UNIX operating system.
AUTH_SHORT	An alternate flavor of AUTH_SYS used by some servers for efficiency. Client programs using AUTH_SYS authentication can receive AUTH_SHORT response verifiers from some servers. See Appendix B, "RPC Protocol and Language Specification" for details.
AUTH_DES	An authentication flavor based on DES encryption techniques.
AUTH_KERB	Version 5 Kerberos authentication based on DES framework.

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

```
clnt = clnt_create(host, prognum, versnum, nettype);
```

the appropriate client-creation routine sets the associated authentication handle to:

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

If you create a new instance of authentication, you must destroy it with `auth_destroy(clnt->cl_auth)`. This destruction conserves memory.

On the server side, the RPC package passes a request that has an arbitrary authentication style associated with it to the service-dispatch routine. The request handle passed to a service-

dispatch routine contains the structure `rq_cred`. This structure is opaque, except for one field: the flavor of the authentication credentials.

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

The RPC package guarantees the following structural requirements to the service-dispatch routine:

- The `rq_cred` field in the `svc_req` structure is well formed. You can check `rq_cred.oa_flavor` to get the flavor of authentication. You can also check the other fields of `rq_cred` if RPC does not support the flavor.

- The `rq_clntcred` field that is passed to service procedures is either `NULL` or points to a well-formed structure that corresponds to a supported flavor of authentication credential. No authentication data exists for the `AUTH_NONE` flavor. *rq_clntcred* can be cast only as a pointer to an `authsys_parms`, `short_hand_verf`, `authkerb_cred`, or `authdes_cred` structure.

AUTH_SYS Authentication

The client can use `AUTH_SYS` style authentication (called `AUTH_UNIX` in previous releases) by setting `clnt–>cl_auth` after creating the RPC client handle:

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

This setting causes each RPC call associated with `clnt` to carry with it the following credentials-authentication structure shown in the following example.

EXAMPLE 56   AUTH_SYS Credential Structure

```
/*
 * AUTH_SYS flavor credentials.
 */
struct authsys_parms {
 u_long aup_time;  /* credentials creation time */
 char *aup_machname;  /* client's host name */
 uid_t aup_uid;   /* client's effective uid */
 gid_t aup_gid;   /* client's current group id */
 u_int aup_len;   /* element length of aup_gids*/
 gid_t *aup_gids;  /* array of groups user is in */
```

```
};
```

`rpc.broadcast` defaults to `AUTH_SYS` authentication.

The following example shows a server, with procedure `RUSERPROC_1()`, that returns the number of users on the network. As an example of authentication, the server checks `AUTH_SYS` credentials and does not service requests from callers with a *uid* of 16.

EXAMPLE 57   Authentication Server

```
nuser(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 struct authsys_parms *sys_cred;
 uid_t uid;
 unsigned int nusers;

 /* NULLPROC should never be authenticated */
 if (rqstp->rq_proc == NULLPROC) {
  if (!svc_sendreply( transp, xdr_void, (caddr_t) NULL))
   fprintf(stderr, "can't reply to RPC call\n");
  return;
 }

 /* now get the uid */
 switch(rqstp->rq_cred.oa_flavor) {
  case AUTH_SYS:
   sys_cred = (struct authsys_parms *) rqstp->rq_clntcred;
   uid = sys_cred->aup_uid;
   break;
  default:
   svcerr_weakauth(transp);
   return;
 }
 switch(rqstp->rq_proc) {
  case RUSERSPROC_1:
   /* make sure caller is allowed to call this proc */
   if (uid == 16) {
    svcerr_systemerr(transp);

    return;
   }
   /*
    * Code here to compute the number of users and assign
    * it to the variable nusers
    */
   if (!svc_sendreply( transp, xdr_u_int, &nusers))
    fprintf(stderr, "can't reply to RPC call\n");
   return;
```

```
  default:
   svcerr_noproc(transp);
   return;
 }
}
```

Note the following points about the example:

■ The authentication parameters associated with the NULLPROC (procedure number zero) are usually not checked.

■ The server calls svcerr_weakauth() if the authentication parameter's flavor is too weak. In this case, there is no way to get the list of authentication flavors the server requires.

■ The service protocol should return status for access denied. In the examples, the protocol instead calls the service primitive svcerr_systemerr().

The last point underscores the relation between the RPC authentication package and the services: RPC deals only with authentication and not with an individual service's access control. The services must establish access-control policies and reflect these policies as return statuses in their protocols.

AUTH_DES Authentication

Use AUTH_DES authentication for programs that require more security than AUTH_SYS provides. AUTH_SYS authentication is easy to defeat. For example, instead of using authsys_create_default(), a program can call authsys_create() and change the RPC authentication handle to give itself any desired user ID and host name.

AUTH_DES authentication requires keyserv() daemons to be running on both the server and client hosts. The NIS naming service must also be running. Users on these hosts need public/secret key pairs assigned by the network administrator in the publickey() database. The users must also have decrypted their secret keys with the keylogin() command. This decryption is normally done by login() unless the login password and secure-RPC password differ.

To use AUTH_DES authentication, a client must set its authentication handle appropriately, as shown in the following example.

```
cl->cl_auth = authdes_seccreate(servername, 60, server,
             (char *)NULL);
```

The first argument is the network name or "net name" of the owner of the server process. Server processes are usually root processes, and you can get their net names with the following call;

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

servername points to the receiving string and `server` is the name of the host the server process is running on. If the server process was run by a non-root user, use the call `user2netname()` as follows:

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

`serveruid()` is the user ID of the server process. The last argument of both functions is the name of the domain that contains the server. `NULL` means "use the local domain name."

The second argument of `authdes_seccreate()` is the lifetime (known also as the "window") of the client's credential. In this example, a credential expires 60 seconds after the client makes an RPC call. If a program tries to reuse the credential, the server RPC subsystem recognizes that the credential has expired and does not service the request carrying the expired credential. If any program tries to reuse a credential within its lifetime, the process is rejected, because the server RPC subsystem saves credentials it has seen in the near past and does not serve duplicates.

The third argument of `authdes_seccreate()` is the name of the *timehost* used to synchronize clocks. `AUTH_DES` authentication requires that server and client agree on the time. Example 57, "Authentication Server," on page 102 specifies synchronization with the server. A `(char *)` `NULL` says not to synchronize. Use this syntax only when you are sure that the client and server are already synchronized.

The fourth argument of `authdes_seccreate()` points to a DES encryption key to encrypt timestamps and data. If this argument is `(char *)NULL`, as it is in Example 57, "Authentication Server," on page 102, a random key is chosen. The `ah_key` field of the authentication handle contains the key.

The server side is simpler than the client. The following example shows the server in Example 57, "Authentication Server," on page 102 changed to use `AUTH_DES`.

EXAMPLE 58   AUTH_DES Server

```
#include <rpc/rpc.h>
 ...
nuser(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 struct authdes_cred *des_cred;
 uid_t uid;
 gid_t gid;
 int gidlen;
 gid_t gidlist[10];

 /* NULLPROC should never be authenticated */
 if (rqstp->rq_proc == NULLPROC) {
```

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

The routine netname2user() converts a network name, or "net name" of a user, to a local system ID. It also supplies group IDs, which are not used in this example.

AUTH_KERB Authentication

 Recent releases of the Oracle Solaris OS include support for most client-side features of Kerberos 5 except klogin. AUTH_KERB is conceptually similar to AUTH_DES. The essential difference is that DES passes a network name and a DES-encrypted session key, while Kerberos passes the encrypted service ticket. This section describes other factors that affect implementation and interoperability.

Kerberos uses the concept of a time window in which its credentials are valid. It does not place restrictions on the clocks of the client or server. Specifically, the *window* is passed as an argument to authkerb_seccreate(). The window does not change. If a *timehost* is specified as an argument, the client side gets the time from the *timehost* and alters its timestamp by the difference in time. Various methods of time synchronization are available. See the kerberos_rpc man page for more information.

Kerberos users are identified by a primary name, instance, and realm. The RPC authentication code ignores the realm and instance, while the Kerberos library code does not. The assumption is that user names are the same between client and server. This enables a server to translate a primary name into user identification information. Two forms of well-known names are used (omitting the realm):

- root.*host* represents a privileged user on client *host*.
- *user*.ignored represents the user whose user name is *user*. The instance is ignored.

Kerberos uses cipher block chaining (CBC) mode when sending a full name credential, one that includes the ticket and window, and electronic code book (ECB) mode otherwise. CBC and ECB are two methods of DES encryption. The session key is used as the initial input vector for CBC mode. The following notation means that XDR is used on *object* as a `type`.

```
xdr_type(object)
```

The length in the next code section is the size, in bytes of the credential or verifier, rounded up to 4-byte units. The full name credential and verifier are obtained as follows:

```
xdr_long(timestamp.seconds)
xdr_long(timestamp.useconds)
xdr_long(window)
xdr_long(window - 1)
```

After encryption with CBC with input vector equal to the session key, the output is two DES cipher blocks:

```
CB0
CB1.low
CB1.high
```

The credential is:

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_enum(AKN_FULLNAME)
xdr_bytes(ticket)
xdr_opaque(CB1.high)
```

The verifier is:

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_opaque(CB0)
xdr_opaque(CB1.low)
```

The nickname exchange yields:

```
xdr_long(timestamp.seconds)
xdr_long(timestamp.useconds)
```

The nickname is encrypted with ECB to obtain `ECB0`, and the credential is:

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_enum(AKN_NICKNAME)
xdr_opaque(akc_nickname)
```

The verifier is:

```
xdr_long(AUTH_KERB)
xdr_long(length)
xdr_opaque(ECB0)
xdr_opaque(0)
```

Authentication Using RPCSEC_GSS

A determined snoop can overcome the authentication flavors mentioned previously- `AUTH_SYS`, `AUTH_DES,` and `AUTH_KERB.` For this reason a new networking layer, the Generic Security Standard API, or GSS-API, was added, which RPC programmers can use. The GSS-API framework offers two extra services beyond authentication: integrity and privacy.

- *Integrity.* With the integrity service, the GSS-API uses the underlying mechanism to authenticate messages exchanged between programs. Cryptographic checksums establish:
  - The identity of the data originator to the recipient
  - The identity of the recipient to the originator if mutual authentication is requested
  - The authenticity of the transmitted data itself
- *Privacy.* The privacy service includes the integrity service. In addition, the transmitted data is also encrypted to protect it from any eavesdroppers.

  Because of U.S. export restrictions, the privacy service might not be available to all users.

Note - Currently, the GSS–API is exposed, and certain GSS-API features are visible through RPCSEC_GSS functions. See the *Developer's Guide to Oracle Solaris 11 Security*.

RPCSEC_GSS API

The RPCSEC_GSS security flavor enables ONC RPC applications to maximize the features of GSS-API. RPCSEC_GSS sits "on top" of the GSS-API layer as shown in the following figure.

FIGURE 3   GSS-API and RPCSEC_GSS Security Layers
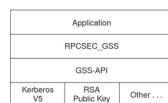
Using the programming interface for RPCSEC_GSS, ONC RPC applications can specify the following information:

- Mechanism – A security paradigm. Each kind of security mechanism offers a different kind of data protection, as well as one or more levels of data protection. You can use any security mechanism supported by the GSS-API (Kerberos V5, RSA public key, and so forth).
- Security service – Either privacy or integrity or neither. The default is integrity. The service is mechanism independent.
- QOP – Quality of protection. The QOP specifies the type of cryptographic algorithm to be used to implement privacy or integrity services. Each security mechanism can have one or more QOPs associated with it.

Applications can obtain lists of valid QOPs and mechanisms through functions provided by RPCSEC_GSS. See "Miscellaneous Functions" on page 116. Developers should avoid hard-coding mechanisms and QOPs into their applications, so that the applications do not need to be modified to use new or different mechanisms and QOPs.

Note - Historically, "security flavor" and "authentication flavor" have had the same meaning. With the introduction of RPCSEC_GSS, "flavor" now has a somewhat different sense. A flavor can now include a service integrity or privacy along with authentication, although currently RPCSEC_GSS is the only flavor that falls into this category.

Using RPCSEC_GSS, ONC RPC applications establish a security context with a peer, exchange data, and destroy the context, just as they do with other flavors. After a context is established, the application can change the QOP and service for each data unit sent.

For more information on RPCSEC_GSS, including RPCSEC_GSS data types, see the `rpcsec_gss(3N)` man page.

RPCSEC_GSS Routines

The following table summarizes RPCSEC_GSS commands. It is a general overview of RPCSEC_GSS functions, rather than a specific description of each one. For more information on each function, see its man page, or check the `rpcsec_gss(3NSL)` man page for an overview, including a list of RPCSEC_GSS data structures.

TABLE 14   RPCSEC_GSS Functions

Action	Function	Input	Output
Create a security context	`rpc_gss_seccreate(3NSL)`	CLIENT handle, principal name, mechanism, QOP, service type	`AUTH` handle
Change QOP, service type for context	`rpc_gss_set_defaults(3NSL)`	Old QOP, service	New QOP, service
Show maximum size for data before security transformation	`rpc_gss_max_data_length(3NSL)` (client side)	Maximum data size allowed by transport	Maximum pre-transformation data size

Action	Function	Input	Output
Show maximum size for data before security transformation	`rpc_gss_svc_max_data_length`(3NSL) (server side)	Maximum data size allowed by transport	Maximum pre-transformation data size
Set name of principals for server to represent	`rpc_gss_set_svc_name`(3NSL)	Principal name, RPC program, version #s	TRUE if successful
Fetch credentials of caller (client)	`rpc_gss_getcred`(3NSL)	Pointer to `svc_req` structure	UNIX credentials, RPCSEC_GSS credentials, cookie
Specify user-written callback function	`rpc_gss_set_callback`(3NSL)	Pointer to callback function	TRUE if successful
Create RPCSEC_GSS structure for principal names from unique parameters	`rpc_gss_get_principal_name`(3NSL)	Mechanism, user name, system name, domain name	RPCSEC_GSS principal name structure
Fetch an error code when an RPCSEC_GSS routine fails	`rpc_gss_get_error`(3NSL)		RPCSEC_GSS error number, `errno` if applicable
Get strings for installed mechanisms	`rpc_gss_get_mechanisms`(3NSL)		List of valid mechanisms
Get valid QOP strings	`rpc_gss_get_mech_info`(3NSL)	Mechanism	Valid QOPs for that mechanism
Get the highest, lowest version numbers of RPCSEC_GSS supported	`rpc_gss_get_versions`(3NSL)		Highest, lowest versions
Check if a mechanism is installed	`rpc_gss_is_installed`(3NSL)	Mechanism	TRUE if installed
Convert ASCII mechanism to RPC object identifier	`rpc_gss_mech_to_oid`(3NSL)	Mechanism (as string)	Mechanism (as OID)
Convert ASCII QOP to integer	`rpc_gss_qop_to_num`(3NSL)	QOP (as string)	QOP (as integer)

Creating a Context

You create contexts with the `rpc_gss_seccreate()` call. This function takes as its arguments:

- A client handle returned, for example, by `clnt_create()`
- The name of the server principal, for example, `nfs@example.com`

- The mechanism (for example, Kerberos V5) for the session
- The security service type (for example, privacy)
- The QOP for the session
- Two GSS-API parameters that can remain opaque for most uses (that is, the programmer can supply NULL values)

This function returns an AUTH authentication handle. The following example shows how `rpc_gss_seccreate()` might be used to create a context using the Kerberos V5 security mechanism and the integrity service.

EXAMPLE 59   rpc_gss_seccreate()

```
CLIENT *clnt;                       /* client handle */
char server_host[] = "foo";
char service_name[] = "nfs@eng.example.com";
char mech[] = "kerberos_v5";

clnt = clnt_create(server_host, SERVER_PROG, SERV_VERS, "netpath");
clnt->clnt_auth = rpc_gss_seccreate(clnt, service_name, mech,
                        rpc_gss_svc_integrity, NULL, NULL, NULL);

. . .
```

Note the following points about the example:

- Although the mechanism was declared explicitly for ease of reading, it would be more commonly obtained programmatically with `rpc_gss_get_mechanisms()` from a table of available mechanisms.
- The QOP is passed as a NULL, which sets the QOP to this mechanism's default. Otherwise, a valid value could, as with the mechanism, be obtained programmatically with `rpc_gss_get_mechanisms()`. See the rpc_gss_get_mechanisms(3NSL) man page for more information.
- The security service type, `rpc_gss_svc_integrity`, is an enum of the RPCSEC_GSS type `rpc_gss_service_t`. `rpc_gss_service_t` has the following format:

```
typedef enum {
    rpc_gss_svc_default = 0,
    rpc_gss_svc_none = 1,
    rpc_gss_svc_integrity = 2,
    rpc_gss_svc_privacy = 3
}  rpc_gss_service_t;
```

  The default security service maps to integrity, so the programmer could have specified `rpc_gss_svc_default` and obtained the same result.

For more information, see the rpc_gss_seccreate(3NSL) man page.

Changing Values and Destroying a Context

After a context has been set, the application might need to change QOP and service values for individual data units being transmitted. For example, if you want a program to encrypt a password but not a login name, you can use `rpc_gss_set_defaults()`.

EXAMPLE 60 `rpc_gss_set_defaults()`

```
rpc_gss_set_defaults(clnt->clnt_auth, rpc_gss_svc_privacy, qop);

. . .
```

In this case, the security service is set to privacy. See "Creating a Context" on page 109. `qop` is a pointer to a string naming the new QOP.

Contexts are destroyed in the usual way, with `auth_destroy()`.

For more information on changing service and QOP, see the `rpc_gss_set_defaults`(3NSL) man page.

Principal Names

You need both a client and a server principal name to establish and maintain a security context.

- A server's principal name is always specified as a NULL-terminated ASCII string of the form *service@host*. One example is `nfs@eng.example.com`.

  When a client creates a security context, it specifies the server principal name in this format. See "Creating a Context" on page 109. Similarly, when a server needs to set the name of a principal it represents, it uses `rpc_gss_set_svc_name()`. This function takes a principal name in this format as an argument.
- The principal name of a client, as received by a server, takes the form of an `rpc_gss_principal_t` structure: a counted, opaque byte string determined by the mechanism being used. This structure is described in the `rpcsec_gss`(3NSL) man page.

Setting Server Principal Names

A server needs to be told the names of the principals it represents when it starts up. A server can act as more than one principal. `rpc_gss_set_svc_name()` sets the name of the principals, as shown in the following code example.

EXAMPLE 61 `rpc_gss_set_svc_name()`

```
char *principal, *mechanism;
```

```
u_int req_time;

principal = "nfs@eng.example.com";
mechanism = "kerberos_v5";
req_time = 10000;  /* time for which credential should be valid */

rpc_gss_set_svc_name(principal, mechanism, req_time, SERV_PROG, SERV_VERS);
```

Kerberos ignores the `req_time` parameter. Other authentication systems might use it.

For more information, see the `rpc_gss_set_svc_name`(3NSL) man page.

Generating Client Principal Names

Servers need to be able to operate on a client's principal name. For example, you might need
to compare a client's principal name to an access control list, or look up a UNIX credential
for that client, if such a credential exists. Such principal names are kept in the form of a
`rpc_gss_principal_t` structure pointer. See the `rpcsec_gss`(3NSL) man page for more on
`rpc_gss_principal_t`. If a server is to compare a principal name it has received with the name
of a known entity, the server needs to be able to generate a principal name in that form.

The `rpc_gss_get_principal_name()` call takes as input several parameters that uniquely
identify an individual on a network, and generates a principal name as a `rpc_gss_principal_t`
structure pointer, as shown in the following code example.

EXAMPLE 62   `rpc_gss_get_principal_name()`

```
rpc_gss_principal_t *principal;
rpc_gss_get_principal_name(principal, mechanism, name, node, domain);
. . .
```

The arguments to `rpc_gss_get_principal_name()` are:

- `principal` is a pointer to the `rpc_gss_principal_t` structure to be set.
- `mechanism` is the security mechanism being used. The principal name being generated is
  mechanism dependent.
- `name` is an individual or service name, such as `joeh` or `nfs`, or even the name of a user-
  defined application.
- `node` might be, for example, a UNIX system name.
- `domain` might be, for example, a DNS, NIS, or a Kerberos realm.

Each security mechanism requires different identifying parameters. For example, Kerberos V5
requires a user name and, only optionally, qualified node and domain names, which in Kerberos
terms are host and realm names.

For more information, see the `rpc_gss_get_principal_name`(3NSL) man page.

Freeing Principal Names

Use the `free()` library call to free principal names.

Receiving Credentials at the Server

A server must be able to fetch the credentials of a client. The `rpc_gss_getcred()` function, shown in Example 63, "Getting Credentials," on page 113, enables the server to retrieve either UNIX credentials or RPCSEC_GSS credentials, or both. The function has two arguments that are set if the function is successful. One is a pointer to an `rpc_gss_ucred_t` structure, which contains the caller's UNIX credentials, if such exist:

```
typedef struct {
    uid_t   uid;         /* user ID */
    gid_t   gid;         /* group ID */
    short   gidlen;
    git_t   *gidlist;    /* list of groups */
} rpc_gss_ucred_t;
```

The other argument is a pointer to a `rpc_gss_raw_cred_t` structure, which looks like this:

```
typedef struct {
 u_int  version;                 /*RPCSEC_GS program version *mechanism;
         char          *qop;
 rpc_gss_principal_t client_principal;  /* client principal name */
 char                 *svc_principal; /*server principal name */
 rpc_gss_service_t   service;       /* privacy, integrity enum */
} rpc_gss_rawcred_t;
```

Because `rpc_gss_rawcred_t` contains both the client and server principal names, `rpc_gss_getcred()` can return them both. See "Generating Client Principal Names" on page 112 for a description of the `rpc_gss_principal_t` structure and how it is created.

The following example is a simple server-side dispatch procedure, in which the server gets the credentials for the caller. The procedure gets the caller's UNIX credentials and then verifies the user's identity, using the mechanism, QOP, and service type found in the `rpc_gss_rcred_t` argument.

EXAMPLE 63 Getting Credentials

```
static void server_prog(struct svc_req *rqstp, SVCXPRT *xprt)
{
  rpc_gss_ucred_t *ucred;
  rpc_gss_rawcred_t *rcred;
```

```
  if (rqst->rq_proq == NULLPROC) {
   svc_sendreply(xprt, xdr_void, NULL);
   return;
  }
  /*
   * authenticate all other requests */
   */

  switch (rqstp->rq_cred.oa_flavor) {
  case RPCSEC_GSS:
   /*
    * get credential information
    */
   rpc_gss_getcred(rqstp, &rcred, &ucred, NULL);
   /*
   * verify that the user is allowed to access
   * using received security parameters by
   * peeking into my config file
   */
   if (!authenticate_user(ucred->uid, rcred->mechanism,
    rcred->qop, rcred->service)) {
    svcerr_weakauth(xprt);
    return;
   }
   break;  /* allow the user in */
  default:
   svcerr_weakauth(xprt);
   return;
  } /* end switch */

  switch (rqstp->rq_proq) {
  case SERV_PROC1:
   . . .
  }

  /* usual request processing; send response ... */

  return;

}
```

For more information, see the rpc_gss_getcred(3NSL) man page.

Cookies

In Example 63, "Getting Credentials," on page 113, the last argument to rpc_gss_getcred()
(here, a NULL) is a user-defined cookie, with a value on return of whatever was specified by
the server when the context was created. This cookie, a 4-byte value, can be used in any way
appropriate for the application. RPC does not interpret the cookie. For example, the cookie can

be a pointer or index to a structure that represents the context initiator. Instead of computing this value for every request, the server computes it at context-creation time, saving on request-processing time.

Callbacks

Another opportunity to use cookies is with callbacks. By using the `rpc_gss_set_callback()` function, a server can specify a user-defined callback so that it knows when a context first gets used. The callback is invoked the first time a context is used for data exchanges, after the context is established for the specified program and version.

The user-defined callback routine takes the following form:

```
bool_t callback (struct svc_req *req, gss_cred_id_t deleg,
gss_ctx_id_t gss_context rpc_gss_lock_t *
lock void ** cookie);
```

The second and third arguments, `deleg` and `gss_context`, are GSS-API data types and are currently exposed. See the *Developer's Guide to Oracle Solaris 11 Security* for more information. Note that `deleg` is the identity of any delegated peer, while `gss_context` is a pointer to the GSS-API context. This pointer is necessary in case the program needs to perform GSS-API operations on the context, that is, to test for acceptance criteria. You have already seen the `cookie` argument.

The `lock` argument is a pointer to a `rpc_gss_lock_t` structure:

```
typedef struct {
  bool_t            locked;
  rpc_gss_rawcred_t   *raw_cred;
} rpc_gss_lock_t;
```

This parameter enables a server to enforce a particular QOP and service for the session. QOP and service are found in the `rpc_gss_rawcred_t` structure described in Example 63, "Getting Credentials," on page 113. A server should not change the values for service and QOP. When the user-defined callback is invoked, the `locked` field is set to `FALSE`. If the server sets `locked` to `TRUE`, only requests with QOP and service values that match the QOP and service values in the `rpc_gss_rawcred_t` structure are accepted.

For more information, see the `rpc_gss_set_callback(3NSL)` man page.

Maximum Data Size

Two functions, `rpc_gss_max_data_length()` on the client side, and `rpc_gss_svc_max_data_length()` on the server side, are useful in determining how large a

piece of data can be before it is transformed by security measures and sent "over the wire." A security transformation such as encryption usually changes the size of a piece of transmitted data, most often enlarging it. To make sure that data won't be enlarged past a usable size, these two functions return the maximum pre-transformation size for a given transport.

For more information, see the `rpc_gss_max_data_length(3NSL)` man page.

Miscellaneous Functions

You can use several functions for getting information about the installed security system.

- `rpc_gss_get_mechanisms(3NSL)()` returns a list of installed security mechanisms.
- `rpc_gss_is_installed(3NSL)()` checks if a specified mechanism is installed.
- `rpc_gss_get_mech_info(3NSL)()` returns valid QOPs for a given mechanism.

Using these functions gives the programmer latitude in avoiding hard-coding security parameters in applications. (See Table 14, "RPCSEC_GSS Functions," on page 108 and the `rpcsec_gss(3NSL)` man page for a list of all RPCSEC_GSS functions.)

Associated Files

RPCSEC_GSS makes use of certain files to store information.

`gsscred` Table

When a server retrieves the client credentials associated with a request, the server can get either the client's principal name in the form of a `rpc_gss_principal_t` structure pointer or local UNIX credentials (UID) for that client. Services such as NFS require a local UNIX credential for access checking, but others might not. Those services can, for example, store the principal name directly in their own access control lists as a `rpc_gss_principal_t` structure.

Note - The correspondence between a client's network credential (its principal name) and any local UNIX credential is not automatic. The local security administrator must be set up explicitly.

The `gsscred` file contains both the client's UNIX and network (for example, Kerberos V5) credentials. The network credential is the Hex-ASCII representation of the `rpc_gss_principal_t` structure. The `gsscred` file is accessed through XFN. Thus, this table can be implemented over files, NIS, or any future name service supported by XFN. In the XFN

hierarchy, this table appears as *this_org_unit*/*service*/gsscred. Administrators can maintain the gsscred table with the use of the gsscred utility, which enables adding and deleting of users and mechanisms.

`/etc/gss/qop` and `/etc/gss/mech`

For convenience, RPCSEC_GSS uses string literals for representing mechanisms and quality of protection (QOP) parameters. The underlying mechanisms themselves, however, require mechanisms to be represented as object identifiers and QOPs as 32–bit integers. Additionally, for each mechanism, you need to specify the shared library that implements the services for that mechanism.

The /etc/gss/mech file stores the following information on all installed mechanisms on a system: the mechanism name, in ASCII; the mechanism's OID; the shared library implementing the services provided by this mechanism; and, optionally, the kernel module implementing the service. A sample line might look like this:

```
kerberos_v5   1.2.840.113554.1.2.2    gl/mech_krb5.so gl_kmech_krb5
```

For all mechanisms installed, the /etc/gss/qop file stores all the QOPs supported by each mechanism, both as an ASCII string and as its corresponding 32–bit integer.

Both /etc/gss/mech and /etc/gss/qop are created when security mechanisms are first installed on a given system.

Many of the in-kernel RPC routines use non-string values to represent mechanism and QOP. Therefore, applications can use the rpc_gss_mech_to_oid() and rpc_gss_qop_to_num() functions to get the non-string equivalents for these parameters, should they need to maximize use of those in-kernel routines.

Using Port Monitors

RPC servers can be started by port monitors such as inetd and listen. Port monitors listen for requests and spawn servers in response. The forked server process is passed the file descriptor 0 on which the request has been accepted. For inetd, when the server is done, it can exit immediately or wait a given interval for another service request.

For listen, servers should exit immediately after replying because listen() always spawns a new process. The following function call creates a SVCXPRT handle to be used by the services started by port monitors.

```
transp = svc_tli_create(0, nconf, (struct t_bind *)NULL, 0, 0)
```

nconf is the `netconfig` structure of the transport from which the request is received.

Because the port monitors have already registered the service with `rpcbind`, the service does not need to register with `rpcbind`. The service must call `svc_reg()` to register the service procedure.

```
svc_reg(transp, PROGNUM, VERSNUM, dispatch,(struct netconfig *)NULL)
```

The `netconfig` structure is `NULL` to prevent `svc_reg()` from registering the service with `rpcbind`.

For connection-oriented transports, the following routine provides a lower level interface:

```
transp = svc_fd_create(0, recvsize, sendsize);
```

A `0` file descriptor is the first argument. You can set the value of *recvsize* and *sendsize* to any appropriate buffer size. A `0` for either argument causes a system default size to be chosen. Application servers that do not do any listening of their own use `svc_fd_create()`.

Using `inetd`

Entries in `/etc/inet/inetd.conf` have different formats for socket-based, TLI-based, and RPC services. The format of `inetd.conf` entries for RPC services follows.

TABLE 15   RPC `inetd` Services

Service	Description
rpc_prog/vers	The name of an RPC program followed by a / and the version number or a range of version numbers.
endpoint_type	One of `dgram` (for connectionless sockets), `stream` (for connection mode sockets), or `tli` (for TLI endpoints).
proto	May be * (for all supported transports), a net type, a net ID, or a comma separated list of net type and net ID.
flags	Either `wait` or `nowait`.
user	Must exist in the effective passwd database.
pathname	Full path name of the server daemon.
args	Arguments to be passed to the daemon on invocation.

For example:

```
rquotad/1 tli rpc/udp wait root /usr/lib/nfs/rquotad rquotad
```

For more information, see the `inetd.conf(4)` man page.

Using the Listener

Use `pmadm` to add RPC services:

```
pmadm -a -p pm_tag -s svctag -i id -v vers \
  -m `nlsadmin -c command -D -R prog:vers`
```

The arguments are:

`-a`	adds a service
`-p` *pm_tag*	specifies a tag associated with the port monitor providing access to the service
`-s` *svctag*	server's identifying code
`-i` *id*	the `/etc/passwd` user name assigned to service *svctag*
`-v` *ver*	the version number for the port monitor's database file
`-m`	specifies the `nlsadmin` command to invoke the service. `nlsadmin` can have additional arguments. For example, to add version `1` of a remote program server named `rusersd`, a `pmadm` command would be:

```
# pmadm -a -p tcp -s rusers -i root -v 4 \
    -m `nlsadmin -c /usr/sbin/rpc.ruserd -D -R 100002:1`
```

The command is given `root` permissions, installed in version `4` of the `listener` database file, and is made available over TCP transports. Because of the complexity of the arguments and options to `pmadm`, use a command script or the menu system to add RPC services. To use the menu system, type `sysadm ports` and choose the `-port_services` option.

After adding a service, the `listener` must be re-initialized before the service is available. To do this, stop and restart the listener, as follows. `rpcbind` must be running.

```
# sacadm -k -p pmtag
# sacadm -s -p pmtag
```

Multiple Server Versions

By convention, the first version number of a program, `PROG`, is named `PROGVERS_ORIG` and the most recent version is named `PROGVERS`. Program version numbers must be assigned consecutively. Leaving a gap in the program version sequence can cause the search algorithm not to find a matching program version number that is defined.

Only the owner of a program should change version numbers. Adding a version number to a program that you do not own can cause severe problems when the owner increments the version number.

Suppose a new version of the `ruser` program returns an `unsigned short` rather than an `int`. If you name this version `RUSERSVERS_SHORT`, a server that supports both versions would do a double register. Use the same server handle for both registrations.

EXAMPLE 64   Server Handle for Two Versions of Single Routine

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

Both versions can be performed by a single procedure, as shown in the following example.

EXAMPLE 65   Procedure for Two Versions of Single Routine

```
void
nuser(rqstp, transp)
 struct svc_req *rqstp;
 SVCXPRT *transp;
{
 unsigned int 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 assign it to the variable nusers
    */
  switch(rqstp->rq_vers) {
   case RUSERSVERS_ORIG:
    if (! svc_sendreply( transp, xdr_u_int,
                                   &nusers))
     fprintf(stderr, "can't reply to RPC
                                        call\n");
```

```
        break;
      case RUSERSVERS_SHORT:
        nusers2 = nusers;
        if (! svc_sendreply( transp, xdr_u_short,
                                      &nusers2))
          fprintf(stderr, "can't reply to RPC
                                            call\n");
        break;
    }
    default:
      svcerr_noproc(transp);
      return;
    }
    return;
}
```

Multiple Client Versions

Because different hosts can run different versions of RPC servers, a client should be capable of accommodating the variations. For example, one server can run the old version of RUSERSPROG(RUSERSVERS_ORIG) while another server runs the newer version (RUSERSVERS_SHORT).

If the version on a server does not match the version number in the client creation call, clnt_call() fails with an RPCPROGVERSMISMATCH error. You can get the version numbers supported by a server and then create a client handle with the appropriate version number. Use either the routine in the following example, or clnt_create_vers(). See the rpc(3NSL) man page for more details.

EXAMPLE 66   RPC Versions on Client Side

```
main()
{
 enum clnt_stat status;
 u_short num_s;
 u_int num_l;
 struct rpc_err rpcerr;
 int maxvers, minvers;
 CLIENT *clnt;

 clnt = clnt_create("remote", RUSERSPROG, RUSERSVERS_SHORT,
              "datagram_v");
 if (clnt == (CLIENT *) NULL) {
  clnt_pcreateerror("unable to create client handle");
  exit(1);
 }
 to.tv_sec = 10;       /* set the time outs */
```

```
            to.tv_usec = 0;

   status = clnt_call(clnt, RUSERSPROC_NUM, xdr_void,
                       (caddr_t) NULL, xdr_u_short,
                             (caddr_t)&num_s, to);
   if (status == RPC_SUCCESS) { /* Found latest version number */
    printf("num = %d\n", num_s);
    exit(0);
   }
   if (status != RPC_PROGVERSMISMATCH) { /* Some other error */
    clnt_perror(clnt, "rusers");
    exit(1);
   }
   /* This version not supported */
   clnt_geterr(clnt, &rpcerr);
   maxvers = rpcerr.re_vers.high;    /* highest version supported */
   minvers = rpcerr.re_vers.low;     /*lowest version supported */

   if (RUSERSVERS_SHORT < minvers || RUSERSVERS_SHORT > maxvers)
 {
                      /* doesn't meet minimum standards */
    clnt_perror(clnt, "version mismatch");
    exit(1);
   }
   (void) clnt_control(clnt, CLSET_VERSION, RUSERSVERS_ORIG);
   status = clnt_call(clnt, RUSERSPROC_NUM, xdr_void,
                             (caddr_t) NULL, xdr_u_int, (caddr_t)&num_l, to);
   if (status == RPC_SUCCESS)
         /* We found a version number we recognize */
    printf("num = %d\n", num_l);
   else {
    clnt_perror(clnt, "rusers");
    exit(1);
   }
 }
```

Using Transient RPC Program Numbers

Occasionally, an application could use RPC program numbers that are generated dynamically. This technique could be used for implementing callback procedures, for example. In the callback example, a client program typically registers an RPC service using a dynamically generated, or transient, RPC program number. The program then passes this number on to a server along with a request. The server then calls back the client program using the transient RPC program number in order to supply the results.

This mechanism might be necessary if processing the client's request takes an excessive amount of time and the client cannot block, assuming it is single threaded. In this case, the server acknowledges the client's request, and calls back later with the results.

Another use of callbacks is to generate periodic reports from a server. The client makes an RPC call to start the reporting, and the server periodically calls back the client with reports using the transient RPC program number supplied by the client program.

Dynamically generated, or transient, RPC program numbers are in the transient range 0x40000000 - 0x5fffffff. The following routine creates a service based on a transient RPC program for a given transport type. The service handle and the transient RPC program number are returned. The caller supplies the service dispatch routine, the version, and the transport type.

EXAMPLE 67 Transient RPC Program–Server Side

```
SVCXPRT *register_transient_prog(dispatch, program, version, netid)
 void (*dispatch)(); /* service dispatch routine */
 rpcproc_t *program;    /* returned transient RPC number */
 rpcvers_t version;     /* program version */
 char *netid;       /* transport id */
{
 SVCXPRT  *transp;
 struct netconfig *nconf;
 rpcprog_t prognum;
 if ((nconf = getnetconfigent(netid)) == (struct netconfig *)NULL)
  return ((SVCXPRT *)NULL);
 if ((transp = svc_tli_create(RPC_ANYFD, nconf,
   (struct t_bind *)NULL, 0, 0)) == (SVCXPRT *)NULL) {
  freenetconfigent(nconf);
  return ((SVCXPRT *)NULL);
 }
 prognum = 0x40000000;
 while (prognum < 0x60000000 && svc_reg(transp, prognum,
             version, dispatch, nconf) == 0) {
  prognum++;
 }
 freenetconfigent(nconf);
 if (prognum >= 0x60000000) {
  svc_destroy(transp);
  return ((SVCXPRT *)NULL);
 }
 *program = prognum;
 return (transp);
}
```

6

Porting From TS-RPC to TI-RPC

The transport-independent RPC (TI-RPC) routines provide the developer with stratified levels of access to the transport layer. The highest-level routines provide complete abstraction from the transport and provide true transport-independence. Lower levels provide access levels similar to the TI-RPC of previous releases.

This section is an informal guide to porting transport-specific RPC (TS-RPC) applications to TI-RPC. Table 16, "Differences Between TI-RPC and TS-RPC," on page 127 shows the differences between selected routines and their counterparts. For information on porting issues concerning sockets and transport layer interface (TLI), see the *Oracle Solaris 11.3 Programming Interfaces Guide*.

Porting an Application

An application based on either TCP or UDP can run in binary-compatibility mode. For some applications you only recompile and relink all source files. Such applications might use simple RPC calls and use no socket or TCP or UDP specifics.

You might need to edit code and write new code if an application depends on socket semantics or features specific to TCP or UDP. For example, the code might use the format of host addresses or rely on the Berkeley UNIX concept of privileged ports.

Applications that are dependent on the internals of the library or the socket implementation, or applications that depend on specific transport addressing, probably require more effort to port and might require substantial modification.

Benefits of Porting

Some of the benefits of porting are:

- Application transport independence means applications operate over more transports than before.

- Use of new interfaces makes your application more efficient.
- Binary compatibility is less efficient than native mode.

IPv6 Considerations for RPC

IPv6 is the successor of IPv4, the most commonly used layer 2 protocol. IPv6 is also known as IP next generation (IPng). For more information, see *Administering TCP/IP Networks, IPMP, and IP Tunnels in Oracle Solaris 11.3*.

Both IPv4 and IPv6 are available to users. Applications choose which stack to use when using COTS (connection-oriented transport service). They can choose TCP or CLTS (connectionless transport service).

The following figure illustrates a typical RPC application running over an IPv4 or IPv6 protocol stack.

FIGURE   4   RPC Applications
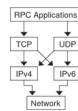
IPv6 is supported only for TI-RPC applications. TS-RPC does not currently support IPv6. Transport selection in TI-RPC is governed either by the `NETPATH` environment variable or in `/etc/netconfig`.

The selection of TCP or UDP instead of IPv4 or IPv6 is dependent on the order in which the corresponding entries appear in `/etc/netconfig`. Two new entries are associated with IPv6 in `/etc/netconfig`, and by default they are the first two entries of the file. TI-RPC first tries IPv6. Failing that, it falls back to IPv4. Doing so requires no change in the RPC application itself provided that it doesn't have any knowledge of the transport and is written using the top-level interface.

Porting Issues

- `libnsl` library – `libc` no longer includes networking functions. `libnsl` must be explicitly specified at compile time to link the network services routines.

- Old interfaces – Many old interfaces are supported in the libnsl library, but they work only with TCP or UDP transports. To make full use of new transports, you must use the new interfaces.
- Name-to-address mapping – Transport independence requires opaque addressing. This requirement has implications for applications that interpret addresses.

Differences Between TI-RPC and TS-RPC

The major differences between transport-independent RPC and transport-specific RPC are illustrated in the following table. Also see "Comparison Examples" on page 130 for code examples comparing TS-RPC with TI-RPC.

TABLE 16   Differences Between TI-RPC and TS-RPC

Category	TI-RPC	TS-RPC
Default Transport Selection	TI-RPC uses the TLI interface.	TS-RPC uses the socket interface.
RPC Address Binding	TI-RPC uses `rpcbind()` for service binding. `rpcbind()` keeps address in universal address format.	TS-RPC uses `portmap` for service binding.
Transport Information	Transport information is kept in a local file, `/etc/netconfig`. Any transport identified in `netconfig` is accessible.	Only TCP and UDP transports are supported.
Loopback Transports	`rpcbind` service requires a secure loopback transport for server registration.	TS-RPC services do not require a loopback transport.
Host Name Resolution	The order of host-name resolution in TI-RPC depends on the order of dynamic libraries identified by entries in `/etc/netconfig`.	Host-name resolution is done by name services. The order is set by the state of the `hosts` database.
File Descriptors	Descriptors are assumed to be TLI endpoints.	Descriptors are assumed to be sockets.
Libraries	TI-RPC requires that applications be linked to the libnsl library.	All TS-RPC functionality is provided in libc.
MT Support	Multithreaded RPC clients and servers are supported.	Multithreaded RPC is not supported.

Function Compatibility Lists

This section lists the RPC library functions and groups them into functional areas. Each section includes lists of functions that are unchanged, have added functionality, and are new to this release.

> **Note -** Functions marked with an asterisk are retained for ease of porting.

Creating and Destroying Services

The following functions are unchanged from the previous releases and are available in the current Oracle Solaris release :

```
svc_destroy
svcfd_create
*svc_raw_create
*svc_tp_create
*svcudp_create
*svc_udp_bufcreate
svc_create
svc_dg_create
svc_fd_create
svc_raw_create
svc_tli_create
svc_tp_create
svc_vc_create
```

Registering and Unregistering Services

The following functions are unchanged from the previous releases and are available in the current Oracle Solaris release:

```
*registerrpc
*svc_register
*svc_unregister
xprt_register
xprt_unregister
rpc_reg
svc_reg
svc_unreg
```

Compatibility Calls

The following functions are unchanged from previous releases and are available in the current Oracle Solaris release:

```
*callrpc
clnt_call
*svc_getcaller - works only with IP-based transports
rpc_call
```

```
svc_getrpccaller
```

Broadcasting

The `clnt_broadcast` call has the same functionality as in previous releases, although it is supported for backward compatibility only.

`clnt_broadcast()` can broadcast only to the `portmap` service. It does not support `rpcbind`.

The `rpc_broadcast` function broadcasts to both `portmap` and `rpcbind` and is also available in the current Oracle Solaris release.

Address Management Functions

The TI-RPC library functions interface with either `portmap` or `rpcbind`. Because the services of the programs differ, there are two sets of functions, one for each service.

The following functions work with `portmap`:

```
pmap_set
pmap_unset
pmap_getport
pmap_getmaps
pmap_rmtcall
```

The following functions work with `rpcbind`:

```
rpcb_set
rpcb_unset
rpcb_getaddr
rpcb_getmaps
rpcb_rmtcall
```

Authentication Functions

The following calls have the same functionality as in previous releases. They are supported for backward compatibility only.

```
authdes_create
authunix_create
authunix_create_default
authdes_seccreate
authsys_create
authsys_create_default
```

Other Functions

`rpcbind` provides a time service, primarily for use by secure RPC client-server time synchronization, available through the `rpcb_gettime()` function. `pmap_getport()` and `rpcb_getaddr()` can be used to get the port number of a registered service. `rpcb_getaddr()` communicates with any server running version 2, 3, or 4 of `rcpbind`. `pmap_getport()` can only communicate with version 2.

Comparison Examples

The changes in client creation from TS-RPC to TI-RPC are illustrated in Example 68, "Client Creation in TS-RPC," on page 130 and Example 69, "Client Creation in TI-RPC," on page 131. Each example:

- Creates a UDP descriptor
- Contacts the remote host's RPC binding process to get the service's address
- Binds the remote service's address to the descriptor
- Creates the client handle and set its timeout

EXAMPLE 68   Client Creation in TS-RPC

```
struct hostent *h;
struct sockaddr_in sin;
int sock = RPC_ANYSOCK;
u_short port;
struct timeval wait;

if ((h = gethostbyname( "host" )) == (struct hostent *) NULL)
{
 syslog(LOG_ERR, "gethostbyname failed");
 exit(1);
}
sin.sin_addr.s_addr = *(u_int *) hp->h_addr;
if ((port = pmap_getport(&sin, PROGRAM, VERSION, "udp")) == 0) {
 syslog (LOG_ERR, "pmap_getport failed");
 exit(1);
} else
 sin.sin_port = htons(port);
wait.tv_sec = 25;
wait.tv_usec = 0;
clntudp_create(&sin, PROGRAM, VERSION, wait, &sock);
```

The TI-RPC version of client creation, shown in the following example, assumes that the UDP transport has the `netid` *udp*. A `netid` is not necessarily a well-known name.

EXAMPLE 69   Client Creation in TI-RPC

```
struct netconfig *nconf;
struct netconfig *getnetconfigent();
struct t_bind *tbind;
struct timeval wait;

nconf = getnetconfigent("udp");
if (nconf == (struct netconfig *) NULL) {
 syslog(LOG_ERR, "getnetconfigent for udp failed");
 exit(1);
}
fd = t_open(nconf->nc_device, O_RDWR, (struct t_info *)NULL);
if (fd == -1) {
 syslog(LOG_ERR, "t_open failed");
 exit(1);
}
tbind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
if (tbind == (struct t_bind *) NULL) {
 syslog(LOG_ERR, "t_bind failed");
 exit(1);
}
if (rpcb_getaddr( PROGRAM, VERSION, nconf, &tbind->addr, "host")
        == FALSE) {
 syslog(LOG_ERR, "rpcb_getaddr failed");
 exit(1);
}
cl = clnt_tli_create(fd, nconf, &tbind->addr, PROGRAM, VERSION,
                        0, 0);
(void) t_free((char *) tbind, T_BIND);
if (cl == (CLIENT *) NULL) {
 syslog(LOG_ERR, "clnt_tli_create failed");
 exit(1);
}
wait.tv_sec = 25;
wait.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, (char *) &wait);
```

Example 70, "Broadcast in TS-RPC," on page 131 and Example 71, "Broadcast in TI-RPC,"
on page 132 show the differences between broadcast in TS-RPC and TI-RPC. The older
clnt_broadcast() is similar to the newer rpc_broadcast(). The primary difference is in the
collectnames() function: it deletes duplicate addresses and displays the names of hosts that
reply to the broadcast.

EXAMPLE 70   Broadcast in TS-RPC

```
statstime sw;
extern int collectnames();

clnt_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
```

```
        xdr_void, NULL, xdr_statstime, &sw, collectnames);
 ...
collectnames(resultsp, raddrp)
 char *resultsp;
 struct sockaddr_in *raddrp;
{
 u_int addr;
 struct entry *entryp, *lim;
 struct hostent *hp;
 extern int curentry;

 /* weed out duplicates */
 addr = raddrp->sin_addr.s_addr;
 lim = entry + curentry;
 for (entryp = entry; entryp < lim; entryp++)
  if (addr == entryp->addr)
    return (0);
...
 /* print the host's name (if possible) or address */
 hp = gethostbyaddr(&raddrp->sin_addr.s_addr, sizeof(u_int),
     AF_INET);
 if( hp == (struct hostent *) NULL)
  printf("0x%x", addr);
 else
  printf("%s", hp->h_name);
}
```

The following code example shows broadcast in TI-RPC.

EXAMPLE 71 Broadcast in TI-RPC

```
statstime sw;
extern int collectnames();

rpc_broadcast(RSTATPROG, RSTATVERS_TIME, RSTATPROC_STATS,
     xdr_void, NULL, xdr_statstime, &sw, collectnames, (char *)
0);
 ...

collectnames(resultsp, taddr, nconf)
 char *resultsp;
 struct t_bind *taddr;
 struct netconfig *nconf;
{
 struct entry *entryp, *lim;
 struct nd_hostservlist *hs;
 extern int curentry;
 extern int netbufeq();

 /* weed out duplicates */
 lim = entry + curentry;
```

```
    for (entryp = entry; entryp < lim; entryp++)
     if (netbufeq( &taddr->addr, entryp->addr))
      return (0);
    ...
    /* print the host's name (if possible) or address */
    if (netdir_getbyaddr( nconf, &hs, &taddr->addr ) == ND_OK)
     printf("%s", hs->h_hostservs->h_host);
    else {
     char *uaddr = taddr2uaddr(nconf, &taddr->addr);
     if (uaddr) {
      printf("%s\n", uaddr);
      (void) free(uaddr);
     } else
      printf("unknown");
    }
}
netbufeq(a, b)
 struct netbuf *a, *b;
{
 return(a->len == b->len && !memcmp( a->buf, b->buf, a->len));
}
```

7

Multithreaded RPC Programming

This manual does not cover basic topics and code examples for the Oracle Solaris implementation of multithreaded programming. Instead, refer to the *Multithreaded Programming Guide* for background on the following topics.

- Thread creation
- Scheduling
- Synchronization
- Signals
- Process resources
- Lightweight processes (LWP)
- Concurrency
- Data-locking strategies

TI-RPC supports multithreaded RPC servers. The difference between a multithreaded server and a single-threaded server is that a multithreaded server uses threading technology to process incoming client requests concurrently. Multithreaded servers can have higher performance and availability compared with single-threaded servers.

MT Client Overview

In a multithread client program, a thread can be created to issue each RPC request. When multiple threads share the same client handle, only one thread at a time is able to make an RPC request. All other threads wait until the outstanding request is complete. On the other hand, when multiple threads make RPC requests using different client handles, the requests are carried out concurrently. Figure 4–1 illustrates a possible timing of a multithreaded client implementation consisting of two client threads using different client handles.

The following figure shows the client side implementation of a multithreaded `rstat` program. The client program creates a thread for each host. Each thread creates its own client handle and makes various RPC calls to the given host. Because the client threads are using different handles to make the RPC calls, they can carry out the RPC calls concurrently.

FIGURE 5   Two Client Threads Using Different Client Handles (Real Time)
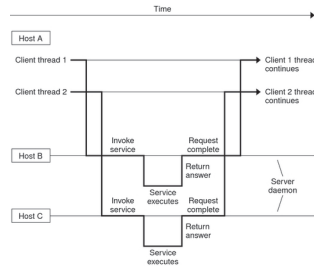
Note - You must link in the thread library when writing any RPC multithreaded-safe application. The thread library must be the last named library on the link line. To link this properly, specify the `-lthread` option in the `compile` command.

Compile the program in the code example by typing `cc rstat.c -lnsl -lthread`.

MT Server Overview

RPC servers made available prior to the Solaris 2.4 release are single threaded. That is, they process client requests sequentially, as the requests come in. For example, say two requests come in, and the first takes 30 seconds to process, and the second takes only 1 second to process. The client that made the second request still has to wait for the first request to complete before it receives a response. This result is not desirable, especially in a multiprocessor server environment, where each CPU could be processing a different request simultaneously.Also, while one request is waiting for I/O to complete, sometimes other requests could be processed by the server.

Facilities in the RPC library for service developers can create multithreaded servers that deliver better performance to end users. Two modes of server multithreading are supported in TI-RPC: the Auto MT mode and the User MT mode.

In the Auto mode, the server automatically creates a new thread for every incoming client request. This thread processes the request, sends a response, and exits. In the User mode, the service developer decides how to create and manage threads for concurrently processing the incoming client requests. The Auto mode is much easier to use than the User mode, but the User mode offers more flexibility for service developers with special requirements.

> **Note -** You must link in the thread library when writing RPC multithreaded-safe applications. The thread library must be the last named library on the link line. To link this properly, specify the `-lthread` option in the compile command.

The two calls that support server multithreading are `rpc_control()` and `svc_done()`. The `rpc_control()` call is used to set the MT mode, either Auto or User mode. If the server uses Auto mode, it does not need to invoke `svc_done()` at all. In User mode, `svc_done()` must be invoked after each client request is processed so that the server can reclaim the resources from processing the request. In addition, multithreaded RPC servers must call on `svc_run()`. Note that `svc_getreqpoll()` and `svc_getreqset()` are unsafe in MT applications.

If the server program does not invoke any of the MT interface calls, it remains in single-threaded mode, which is the default mode.

You are required to make RPC server procedures multithreaded safe regardless of which mode the server is using. Usually, this means that all static and global variables need to be protected with mutex locks. Mutual exclusion and other synchronization APIs are defined in `/usr/include/synch.h` and `/usr/include/pthread.h`.

The following figure illustrates a possible timing of a server implemented in one of the MT modes of operation.

FIGURE   6   MT RPC Server Timing Diagram
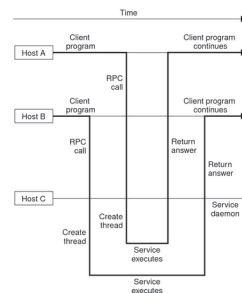
Sharing the Service Transport Handle

The service transport handle, `SVCXPRT`, contains a single data area for decoding arguments and encoding results. Therefore, in the default, single-threaded mode, this structure cannot be freely shared between threads that call functions that perform these operations. However, when a server is operating in the MT Auto or User modes, a copy of this structure is passed to

the service dispatch procedure in order to enable concurrent request processing. Under these circumstances, some routines that would otherwise be unsafe become safe. Unless otherwise noted, the server interfaces are generally MT safe. See the `rpc_svc_calls(3NSL)` man page for more details on safety for server-side interfaces.

MT Auto Mode

In the Auto mode, the RPC library creates and manages threads. The service developer invokes a new interface call, `rpc_control()`, to put the server into MT Auto mode before invoking the `svc_run()` call. In this mode, the programmer needs only to ensure that service procedures are MT safe.

`rpc_control()` enables applications to set and modify global RPC attributes. At present, this function supports only server-side operations. The following table shows the `rpc_control()` operations defined for Auto mode. See also the `rpc_control(3NSL)` man page for additional information.

TABLE 17   `rpc_control()` Library Routines

Routine	Description
`RPC_SVC_MTMODE_SET()`	Set multithread mode
`RPC_SVC_MTMODE_GET()`	Get multithread mode
`RPC_SVC_THRMAX_SET()`	Set maximum number of threads
`RPC_SVC_THRMAX_GET()`	Get maximum number of threads
`RPC_SVC_THRTOTAL_GET()`	Total number of threads currently active
`RPC_SVC_THRCREATES_GET()`	Cumulative total number of threads created by the RPC library
`RPC_SVC_THRERRORS_GET()`	Number of `thr_create()` errors within RPC library

Note - All of the get operations in Table 17, "`rpc_control()` Library Routines," on page 138, except `RPC_SVC_MTMODE_GET()`, apply only to the Auto MT mode. If used in MT User mode or the single-threaded default mode, the results of the operations might be undefined.

By default, the maximum number of threads that the RPC server library creates at any time is 16. If a server needs to process more than 16 client requests concurrently, the maximum number of threads must be set to the desired number. This parameter can be set at any time by the server. It enables the service developer to put an upper bound on the thread resources consumed by the server. Example 72, "Server for MT Auto Mode," on page 139 is an example RPC program written in MT Auto mode. In this case, the maximum number of threads is set at 20.

MT performance is enhanced if the function `svc_getargs()` is called by every procedure other than `NULLPROCS`, even if there are no arguments (you can use `xdr_void()` in this case). This

is true for both the MT Auto and MT User modes. For more information on this call, see the
rpc_svc_calls(3NSL) man page.

Note - You must link in the thread library when writing RPC multithreaded-safe applications.
The thread library must be the last named library on the link line. Specify the -lthread option
in the compile command.

The following example illustrates the server in MT Auto mode. To compile this program, type
cc time_svc.c -lnsl -lthread.

EXAMPLE 72   Server for MT Auto Mode

```
#include <stdio.h>

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

 #include <thread.h>
 #include "time_prot.h"

 void time_prog();

 main(argc, argv)
 int argc;
 char *argv[];
 {
 int transpnum;
 char *nettype;
 int mode = RPC_SVC_MT_AUTO;
 int max = 20;       /* Set maximum number of threads to 20 */

 if (argc > 2) {
  fprintf(stderr, "usage: %s [nettype]\n", argv[0]);
  exit(1);
 }

 if (argc == 2)
  nettype = argv[1];
 else
  nettype = "netpath";

 if (!rpc_control(RPC_SVC_MTMODE_SET, &mode)) {
  printf("RPC_SVC_MTMODE_SET: failed\n");
  exit(1);
 }
 if (!rpc_control(RPC_SVC_THRMAX_SET, &max)) {
  printf("RPC_SVC_THRMAX_SET: failed\n");
  exit(1);
 }
```

```
            transpnum = svc_create( time_prog, TIME_PROG, TIME_VERS,
             nettype);

            if (transpnum == 0) {
             fprintf(stderr, "%s: cannot create %s service.\n",
             argv[0], nettype);
             exit(1);
            }
            svc_run();
           }

           /*
            * The server dispatch function.
            * The RPC server library creates a thread which executes
            * the server dispatcher routine time_prog().  After which
            * the RPC library destroys the thread.
            */

           static void
           time_prog(rqstp, transp)
            struct svc_req *rqstp;
            SVCXPRT *transp;
           {

            switch (rqstp->rq_proc) {
             case NULLPROC:
              svc_sendreply(transp, xdr_void, NULL);
              return;
             case TIME_GET:
              dotime(transp);
              break;
             default:
              svcerr_noproc(transp);
              return;
            }
           }
           dotime(transp)
           SVCXPRT *transp;
           {

            struct timev rslt;
            time_t thetime;

            thetime = time((time_t *)0);
            rslt.second = thetime % 60;
            thetime /= 60;
            rslt.minute = thetime % 60;
            thetime /= 60;
            rslt.hour = thetime % 24;
           if (!svc_sendreply(transp, xdr_timev,(caddr_t) &rslt)) {
             svcerr_systemerr(transp);
```

```
 }
 }
```

The following code example shows the `time_prot.h` header file for the server.

EXAMPLE 73   MT Auto Mode: `time_prot.h`

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


  struct timev {
   int second;

   int minute;
   int hour;
  };


  typedef struct timev timev;

  bool_t xdr_timev();

  #define TIME_PROG 0x40000001

  #define TIME_VERS 1
  #define TIME_GET 1
```

MT User Mode

In MT User mode, the RPC library does not create any threads. This mode works, in principle, like the single-threaded, or default mode. The only difference is that it passes copies of data structures, such as the transport service handle to the service-dispatch routine to be MT safe.

The RPC server developer takes the responsibility for creating and managing threads through the thread library. In the dispatch routine, the service developer can assign the task of procedure execution to newly created or existing threads. The `thr_create()` API is used to create threads having various attributes. All thread library interfaces are defined in `/usr/include/thread.h` and `/usr/include/pthread.h`.

This mode provides flexibility to the service developer. Threads can now have different stack sizes based on service requirements. Threads can be bound. Different procedures can be executed by threads with different characteristics. The service developer might choose to run some services single threaded. The service developer might choose to do special thread-specific signal processing.

As in the Auto mode, you use the `rpc_control()` library call to turn on User mode. Note that the `rpc_control()` operations shown in Table 17, "`rpc_control()` Library Routines," on page

138, except for `RPC_SVC_MTMODE_GET()`, apply only to MT Auto mode. If used in MT User mode or the single-threaded default mode, the results of the operations can be undefined.

Freeing Library Resources in User Mode

In the MT User mode, service procedures must invoke `svc_done()` before returning. `svc_done()` frees resources allocated to service a client request directed to the specified service transport handle. This function is invoked after a client request has been serviced, or after an error or abnormal condition that prevents a reply from being sent. After `svc_done()` is invoked, the service procedure should not reference the service transport handle. The following example shows a server in MT User mode.

Note - `svc_done()` must only be called within MT User mode. For more information on this call, see the `rpc_svc_calls(3NSL)` man page.

EXAMPLE 74   MT User Mode: `rpc_test.h`

```
#define SVC2_PROG 0x30000002
 #define SVC2_VERS 1
 #define SVC2_PROC_ADD 1)
 #define SVC2_PROC_MULT 2

 struct intpair {
  u_short a;
  u_short b;
 };

 typedef struct intpair intpair;

 struct svc2_add_args {
  int argument;
  SVCXPRT *transp;
 };

 struct svc2_mult_args {
  intpair mult_argument;
  SVCXPRT *transp;
 };

 extern bool_t xdr_intpair();

 #define NTHREADS_CONST 500
```

The following code example is the client for MT User mode.

EXAMPLE 75   Client for MT User Mode

```
#define  _REENTRANT
#include <stdio.h>

#include <rpc/rpc.h>
#include <sys/uio.h>

#include <netconfig.h>
#include <netdb.h>

#include <rpc/nettype.h>
#include <thread.h>
#include "rpc_test.h"
void *doclient();
int NTHREADS;
struct thread_info {
 thread_t client_id;
 int client_status;
};
struct thread_info save_thread[NTHREADS_CONST];
main(argc, argv)
 int argc;
 char *argv[];
{
 int index, ret;
 int thread_status;
 thread_t departedid, client_id;
 char *hosts;
 if (argc < 3) {
  printf("Usage: do_operation [n] host\n");
  printf("\twhere n is the number of threads\n");
  exit(1);
 } else
  if (argc == 3) {
   NTHREADS = NTHREADS_CONST;
   hosts = argv[1];  /* live_host */
  } else {
   NTHREADS = atoi(argv[1]);
   hosts = argv[2];
  }
 for (index = 0; index < NTHREADS; index++){
  if (ret = thr_create(NULL, NULL, doclient,
  (void *)  hosts, THR_BOUND, &client_id)){
   printf("thr_create failed: return value %d", ret);
   printf(" for %dth thread\n", index);
   exit(1);
  }
  save_thread[index].client_id = client_id;
 }
 for (index = 0; index < NTHREADS; index++){
```

```
  if (thr_join(save_thread[index].client_id, &departedid,
  (void *)
  &thread_status)){
   printf("thr_join failed for thread %d\n",
   save_thread[index].client_id);
   exit(1);
  }
  save_thread[index].client_status = thread_status;
 }
}
 void *doclient(host)
 char *host;
{
 struct timeval tout;
 enum clnt_stat test;
 int result = 0;
 u_short mult_result = 0;
 int add_arg;
 int EXP_RSLT;
 intpair pair;
 CLIENT *clnt;
 if ((clnt = clnt_create(host, SVC2_PROG, SVC2_VERS, "udp"
==NULL) {
  clnt_pcreateerror("clnt_create error: ");
  thr_exit((void *) -1);
 }
 tout.tv_sec = 25;
 tout.tv_usec = 0;
 memset((char *) &result, 0, sizeof (result));

 memset((char *) &mult_result, 0, sizeof (mult_result));
 if (thr_self() % 2){
  EXP_RSLT = thr_self() + 1;
  add_arg = thr_self();
  test = clnt_call(clnt, SVC2_PROC_ADD, (xdrproc_t) xdr_int,

  (caddr_t) &add_arg, (xdrproc_t) xdr_int, (caddr_t) &result,
  tout);
 } else {
  pair.a = (u_short) thr_self();
  pair.b = (u_short) 1;
  EXP_RSLT = pair.a * pair.b;
  test = clnt_call(clnt, SVC2_PROC_MULT, (xdrproc_t)
  xdr_intpair,
  (caddr_t) &pair, (xdrproc_t) xdr_u_short,

  (caddr_t) &mult_result, tout);
  result = mult_result;
 }
 if (test != RPC_SUCCESS) {
```

```
  printf("THREAD: %d clnt_call hav
  thr_exit((void *) -1);
 };
 thr_exit((void *) 0);
}
```

MT performance is enhanced if the function `svc_getargs()` is called by every procedure other than `NULLPROC`, even if there are no arguments. *xdr_void* may be used in this case. This result is true for both the MT Auto and MT User modes. For more information on this call, see the `rpc_svc_calls(3NSL)` man page.

Note - You must link in the thread library when writing RPC multithreaded-safe applications. The thread library must be the last named library on the link line. Specify the `-lthread` option in the compile command.

♦ ♦ ♦   **C H A P T E R 8**

8

Extensions to the Oracle Solaris RPC Library

New features have been added to the Oracle Solaris RPC library which are integrated into the standard Solaris 9 product.

New and altered man pages are available to describe the functionality added to the Oracle Solaris RPC library.

The additions to the Oracle Solaris RPC library are described in the following sections:

New Features

The new features added to the Oracle Solaris RPC library are:

- One-way messaging - Reduces the time a client thread waits before continuing processing.
- Non-blocking I/O - Enables a client to send requests without being blocked.
- Client connection closure callback - Enables a server to detect client disconnection and to take corrective action.
- Callback user file descriptor - Extends the RPC server to handle non-RPC descriptors.

One-Way Messaging

In one-way messaging the client thread sends a request containing a message to the server. The client thread does not wait for a reply from the server and is free to continue processing when the request has been accepted by the transport layer. The request is not always sent immediately to the server by the transport layer, but waits in a queue until the transport layer sends it. The server executes the request received by processing the message contained in the request. This method saves processing time.

The following figure illustrates one-way messaging.

FIGURE 7   One-Way Messaging
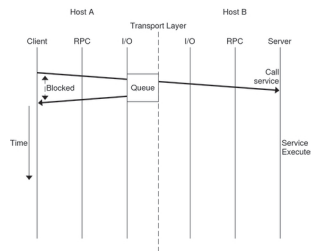
In previous versions of the Oracle Solaris RPC library, most requests were sent by two-way messaging. In two-way messaging, the client thread waits until it gets an answer from the server before continuing processing. If the client thread does not receive a reply from the server within a certain period of time, a time-out occurs. This client thread cannot send a second request until the first request is executed or until a time-out occurs. This messaging method is illustrated in the following figure.

FIGURE 8   Two-Way Messaging
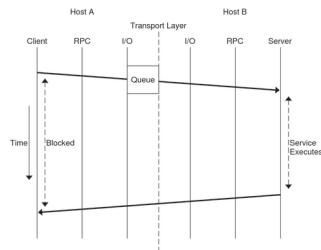
Previous versions of the Oracle Solaris RPC library contain a second method of messaging called batching. In this method, client request are held in a queue until a group of requests can be processed at the same time. This is a form of one-way messaging. See Chapter 4, "Programmer's Interface to RPC" for further details.

After the transport layer accepts a request, the client is not notified of failures in transmission and does not receive a receipt from the server from the request. For example, if the server refuses the request due to an authentication problem, the client is not notified of this problem. If the transport layer does not accept the request, the sending operation returns an immediate error to the client.

If you need to check whether the server is functioning correctly, you can send a two-way request to the server. Such a request can determine whether the server is still available and whether it has received the one-way requests sent by the client.

For one-way messaging, the `clnt_send()` function has been added to the Oracle Solaris RPC library, and the `oneway` attribute has been added to the RPC grammar.

clnt_send()

In previous versions of the Oracle Solaris RPC library, you used the `clnt_call()` function to send a remote procedure call. With the extended one-way messaging service, the `clnt_send()` function sends one-way remote procedure calls.

When the client calls `clnt_send()`, the client sends a request to the server and continues processing. When the request arrives at the server, the server calls a dispatch routine to process the incoming request.

Like `clnt_call()`, the `clnt_send()` function uses a client handle to access a service. See the `clnt_send(3NSL)` and `clnt_call(3NSL)` man pages for further information.

If you do not provide the correct version number to `clnt_create()`, `clnt_call()` fails. In the same circumstances, `clnt_send()` does not report the failure, as the server does not return a status.

oneway Attribute

To use one-way messaging, add the `oneway` keyword to the XDR definition of a server function. When you use the `oneway` keyword, the stubs generated by `rpcgen` use `clnt_send()`. You can either:

- Use a `simplified interface` as outlined in the Chapter 2, "Introduction to TI-RPC". The stubs used by the simplified interface must call `clnt_send()`.
- Call the `clnt_send()` function directly, as described in the `clnt_send(3NSL)` man page.

For one-way messaging, use version 1.1 of the `rpcgen` command.

When declaring the `oneway` keyword, follow the RPC language specification using the following syntax:

```
"oneway" function-ident "(" type-ident-list ")" "=" value;
```

See Appendix B, "RPC Protocol and Language Specification" for details on RPC language specifications.

When you declare the `oneway` attribute for an operation, no result is created on the server side and no message is returned to the client.

The following information on the `oneway` attribute must be added to the RPC Language Definition Table as described in "RPC Language Specification" on page 222:

```
type-ident procedure-ident (type-ident) = value
oneway procedure-ident (type-ident) = value
```

One-way call using a simple counter service

This section describes how to use a one-way procedure on a simple counter service. In this counter service the `ADD()` function is the only function available. Each remote call sends an integer and this integer is added to a global counter managed by the server. For this service, you must declare the `oneway` attribute in the RPC language definition.

In this example, you generate stubs using the `-M`, `-N` and `-C` `rpcgen` options. These options ensure that the stubs are multithread safe, accept multiple input parameters and that generated headers are ANSI C++ compatible. Use these `rpcgen` options even if the client and server applications are mono-threaded as the semantic to pass arguments is clearer and adding threads in applications is easier since the stubs do not change.

1. First, you write the service description in the `counter.x`.

   ```
       /* counter.x: Remote counter protocol */
   program COUNTERPROG {
     version COUNTERVERS {
       oneway ADD(int) = 1;
     } = 1;
   } = 0x20000001;
   ```

   The service has a program number, (COUNTERPROG) 0x200000001, and a version number, (COUNTERVERS) 1.

2. Next, call `rpcgen` on the `counter.x` file.

   ```
   rpcgen -M -N -C counter.x
   ```

   This call generates the client and server stubs, `counter.h`, `counter_clnt.c` and `counter_svc.c`.

3. As shown in the `server.c` file below, write the service handler for the server side and the `counterprog_1_freeresult()` function used to free memory areas allocated to the handler. The RPC library calls this function when the server sends a reply to the client.

   ```
       #include <stdio.h>
   #include "counter.h"
   ```

```
int counter = 0;

    bool_t
add_1_svc(int number, struct svc_req *rqstp)
{
   bool_t retval = TRUE;

    counter = counter + number;

    return retval;
}
   int
counterprog_1_freeresult(SVCXPRT *transp, xdrproc_t xdr_result, caddr_t
                         result)
{
   (void) xdr_free(xdr_result, result);

   /*
   * Insert additional freeing code here, if needed
   */

    return TRUE;
}
```

You build the server by compiling and linking the service handler to the `counter_svc.c`
stub. This stub contains information on the initialization and handling of TI-RPC.

4. Next, you write the client application, `client.c`.

```
    #include <stdio.h>
#include "counter.h"

main(int argc, char *argv[])
{
   CLIENT *clnt;
   enum clnt_stat result;
   char *server;
   int number;

    if(argc !=3) {
     fprintf(stderr, "usage: %s server_name number\n", argv[0];
     exit(1);
    }
   server = argv[1];
   number = atoi(argv[2]);

    /*
```

```
 * Create client handle
 */
clnt = clnt_create(server, COUNTERPROG, COUNTERVERS, "tcp");

 if(clnt == (CLIENT *)NULL) {
/*
 * Couldn't establish connection
 */
   clnt_pcreateerror(server);
  exit(1);
}

 result = add_1(number, clnt);
if (result !=RPC_SUCCESS) {
  clnt_perror(clnt, "call failed");
}

 clnt_destroy(clnt);
  exit(0);
}
```

The add_1() client function is the counter_clnt.c stub generated for the remote function.

To build the client, compile and link the client main and the counter_clnt.c.

5. To launch the server that you built, type ./server

6. Finally, to invoke the service in another shell, type: ./client servername23.

23 is the number being added to the global counter.

Non-Blocking I/O

Non-blocking I/O avoids the client being blocked while waiting for a request to be accepted by the transport layer during one-way messaging for connection-oriented protocols.

For connection-oriented protocols, there is a limit to the amount of data that can be put in a network protocol queue. The limit depends on the transport protocols used. When a client sending a request reaches the data limit, this client is blocked from processing until its request has entered the queue. You cannot determine how long a message will wait before being added to the queue.

In non-blocking I/O, when the transport queue is full, there is an additional buffer available between the client and the transport layer. As requests not accepted by the transport queue can be stored in this buffer, the client is not blocked. The client is free to continue processing as soon as it has put the request in the buffer. The client does not wait until the request is put in the queue and does not receive information on the status of the request after the buffer accepts the request.

By using non-blocking I/O you gain further processing time as compared to two-way and one-way messaging. The client can send requests in succession without being blocked from processing.

The following figure shows a case where you choose the non-blocking argument of the I/O mode and the transport layer queue is full.

FIGURE   9   Non-Blocking Messaging
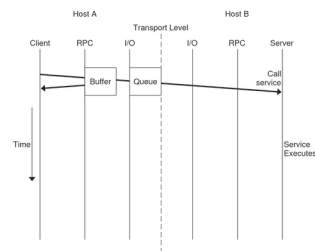
Using Non-Blocking I/O

To use non-blocking I/O, configure a client handle using the CLSET_IO_MODE `rpciomode_t*` option of the `clnt_control()` function with the RPC_CL_NONBLOCKING argument. See the `clnt_control(3NSL)` man page for further information.

When the transport queue is full, the buffer is used. The buffer continues to be used until two criteria are fulfilled:

- The buffer is empty
- The queue can immediately accept a request

Requests then go directly to the transport queue until the queue is full. The default size of the buffer is 16 kbytes.

Note that the buffer is not emptied automatically. You must flush the buffer when it contains data.

When you chose the RPC_CL_NONBLOCKING argument of CLSET_IO_MODE, you have a choice of flush modes. You can specify either the RPC_CLBESTEFFORT_FLUSH or RPC_CL_BLOCKING_FLUSH argument to CLSET_FLUSH_MODE. You can also empty the buffer by sending a synchronous call, such as `clnt_call()`. See the `clnt_control(3NSL)` man page for further information.

If the buffer is full, an `RPC_CANTSTORE` error is returned to the client and the request is not sent. The client must send the message again later. You can find out or change the

size of the buffer by using the CLSET_CONNMAXREC and CLGET_CONNMAXREC commands. To determine the size of all pending request stored in the buffer, use the CLGET_CURRENT_REC_SIZE command. For further information on these commands see the clnt_control(3NSL) man page.

The server does not confirm whether the request is received or processed. After a request enters a buffer, you can use clnt_control() to obtain information on the status of the request.

Using a simple counter with non-blocking I/O

The client.c file in the one-way messaging example is modified in this section to demonstrate how to use the non-blocking I/O mode. In this new file, client_nonblo.c, the I/O mode is specified as non-blocking with the RPC_CL_NONBLOCKING argument, the flush mode is chosen to be blocking by use of the RPC_CL_BLOCKING_FLUSH. The I/O mode and flush mode are invoked with CLSET_IO_MODE. When an error occurs RPC_CANT_STORE is returned to the client and the program tries to flush the buffer. To choose a different method of flush consult the clnt_control(3NSL) man page.

```c
#include <stdio.h>
#include "counter.h"

main(int argc, char *argv[])
{
    CLIENT* clnt;
    enum clnt_stat result;
    char *server;
    int number;
    bool_t bres;
  /*
   * Choose the I/O mode and flush method to be used.
   * The non-blocking I/O mode and blocking flush are
   * chosen in this example.
   */
    int mode = RPC_CL_NONBLOCKING;
    int flushMode = RPC_CL_BLOCKING_FLUSH;

    if (argc != 3) {
     fprintf(stderr, "usage: %s server_name number\n", argv[0]);
     exit(1);
    }
    server = argv[1];
    number = atoi(argv[2]);

    clnt= clnt_create(server, COUNTERPROG, COUNTERVERS, "tcp");
    if (clnt == (CLIENT*) NULL) {
     clnt_pcreateerror(server);
     exit(1);
    }
```

```
/*
 * Use clnt_control to set the I/O mode.
 * The non-blocking I/O mode is
 * chosen for this example.
 */
 bres = clnt_control(clnt, CLSET_IO_MODE, (char*)&mode);
 if (bres)
  /*
  * Set flush mode to blocking
  */
     bres = clnt_control(clnt, CLSET_FLUSH_MODE, (char*)&flushMode);

 if (!bres) {
     clnt_perror(clnt, "clnt_control");
     exit(1);
 }
/*
 * Call the RPC services.
 */
 result = add_1(number, clnt);

 switch (result) {
 case RPC_SUCCESS:
     fprintf(stdout,"Success\n");
     break;
/*
 * RPC_CANTSTORE is a new value returned to the
 * client when the buffer cannot store a request.
 */
 case RPC_CANTSTORE:
     fprintf(stdout,"RPC_CANTSTORE error. Flushing ... \n");
/*
 * The buffer is flushed using the blocking flush method
 */
  bres = clnt_control(clnt, CLFLUSH, NULL);
     if (!bres) {
       clnt_perror(clnt, "clnt_control");
  }
     break;
 default:
  clnt_perror(clnt, "call failed");
     break;
 }

 /* Flush */
 bres = clnt_control(clnt, CLFLUSH, NULL);
 if (!bres) {
  clnt_perror(clnt, "clnt_control");
 }

 clnt_destroy(clnt);
```

```
        exit(0);
}
```

`clnt_call()` Configured as Non-Blocking

For a one-way message, use the `clnt_send()` function. No time-out is applied as the client sends a request to a server and does not wait for a reply.

For two-way messaging, use `clnt_call()`. The client remains blocked until the server sends a reply or an error status message, or until a time-out occurs at the client side.

The non-blocking feature enables you to send two-way and one-way calls together. If you use `clnt_call()` on the client side configured as non-blocking, that is, using the RPC_CL_NONBLOCKING I/O MODE, you get the following modified behavior. When a two-way request is sent to the buffer, all one-way requests already in the buffer are sent through the transport layer before the two-way request is processed. The time taken to empty the buffer is not counted in the two-way call timeout. For further information, see the `clnt_control(3NSL)` man page.

Client Connection Closure Callback

Client connection closure callback enables the server for connection-oriented transport to detect that the client has disconnected. The server can take the necessary action to recover from transport errors. Transport errors occur when a request arrives at the server, or when the server is waiting for a request and the connection is closed.

The connection closure callback is called when no requests are currently being executed on the connection. If the client connection is closed when a request is being executed, the server executes the request but a reply may not be sent to the client. The connection closure callback is called when all pending request are completed.

When a connection closure occurs, the transport layer sends an error message to the client. The handler is attached to a service using `svc_control()` for example as follows:

```
svc_control(service, SVCSET_RECVERRHANDLER, handler);
```

The arguments of `svc_control()` are:

1. A service or an instance of this service. When this argument is a service, any new connection to the service inherits the error handler. When this argument is an instance of the service, only this connection gets the error handler.

2. The error handler callback. The prototype of this callback function is:

```
void handler(const SVCXPRT *svc, const boot_t IsAConnection);
```

For further information see the svc_control(3NSL) man page.

Note - For XDR unmarshalling errors, if the server is unable to unmarshal a request, the message is destroyed and an error is returned directly to the client.

Example of client connection closure callback

This example implements a message log server. A client can use this server to open a log (actually a text file), to store message log, and then to close the log.

The `log.x` file describes the log program interface.

```
enum log_severity { LOG_EMERG=0, LOG_ALERT=1, LOG_CRIT=2, LOG_ERR=3,
     LOG_WARNING=4, LOG_NOTICE=5, LOG_INFO=6 };

program LOG {
     version LOG_VERS1 {
         int OPENLOG(string ident) = 1;

         int CLOSELOG(int logID) = 2;

         oneway WRITELOG(int logID, log_severity severity,
    string message) = 3;
     } = 1;
} = 0x20001971;
```

The two procedures `OPENLOG` and `CLOSELOG` open and close a log that is specified by its logID. The `WRITELOG()` procedure, declared as `oneway` for the example, logs a message in an opened log. A log message contains a severity attribute, and a text message.

This is the makefile for the log server. Use this makefile to call the `log.x` file.

```
RPCGEN = rpcgen

CLIENT = logClient
CLIENT_SRC = logClient.c log_clnt.c log_xdr.c
CLIENT_OBJ = $(CLIENT_SRC:.c=.o)

SERVER = logServer
SERVER_SRC = logServer.c log_svc.c log_xdr.c
SERVER_OBJ = $(SERVER_SRC:.c=.o)

RPCGEN_FILES = log_clnt.c log_svc.c log_xdr.c log.h

CFLAGS += -I.

RPCGEN_FLAGS = -N -C
LIBS = -lsocket -lnsl
```

```
all: log.h ./$(CLIENT) ./$(SERVER)


$(CLIENT): log.h $(CLIENT_OBJ)
    cc -o $(CLIENT) $(LIBS) $(CLIENT_OBJ)

$(SERVER): log.h $(SERVER_OBJ)
    cc -o $(SERVER)  $(LIBS) $(SERVER_OBJ)

$(RPCGEN_FILES): log.x
    $(RPCGEN) $(RPCGEN_FLAGS) log.x

clean:
    rm -f $(CLIENT_OBJ) $(SERVER_OBJ) $(RPCGEN_FILES)
```

`logServer.c` shows the implementation of the log server. As the log server opens a file to store the log messages, it registers a closure connection callback in `openlog_1_svc()`. This callback is used to close the file descriptor even if the client program forgets to call the `closelog()` procedure (or crashes before doing so). This example demonstrates the use of the connection closure callback feature to free up resources associated to a client in an RPC server.

```c
#include "log.h"
#include <stdio.h>
#include <string.h>

#define NR_LOGS 3

typedef struct {
    SVCXPRT* handle;
    FILE* filp;
    char* ident;
} logreg_t;


static logreg_t logreg[NR_LOGS];
static char* severityname[] = {"Emergency", "Alert", "Critical", "Error",
                               "Warning", "Notice", "Information"};

    static void
close_handler(const SVCXPRT* handle, const bool_t);


    static int
get_slot(SVCXPRT* handle)
{
    int i;

    for (i = 0; i < NR_LOGS; ++i) {
        if (handle == logreg[i].handle) return i;
    }
    return -1;
```

```
}

    static FILE*
_openlog(char* logname)
/*
 * Open a log file
 */
{
    FILE* filp = fopen(logname, "a");
    time_t t;

    if (NULL == filp) return NULL;

    time(&t);
    fprintf(filp, "Log opened at %s\n", ctime(&t));

    return filp;
}

    static void
_closelog(FILE* filp)
{
    time_t t;

    time(&t);
    fprintf(filp, "Log close at %s\n", ctime(&t));
  /*
   * Close a log file
   */
    fclose(filp);
}

    int*
openlog_1_svc(char* ident, struct svc_req* req)
{
    int slot = get_slot(NULL);
    FILE* filp;
    static int res;
    time_t t;

    if (-1 != slot) {
        FILE* filp = _openlog(ident);
        if (NULL != filp) {
            logreg[slot].filp = filp;
            logreg[slot].handle = req->rq_xprt;
            logreg[slot].ident = strdup(ident);

  /*
   * When the client calls clnt_destroy, or when the
   * client dies and clnt_destroy is called automatically,
   * the server executes the close_handler callback
```

```
        */
                if (!svc_control(req->rq_xprt, SVCSET_RECVERRHANDLER,
                    (void*)close_handler)) {
            puts("Server: Cannot register a connection closure callback");
            exit(1);
             }
             }

        }
        res = slot;
        return &res;
    }

        int*
    closelog_1_svc(int logid, struct svc_req* req)
    {
        static int res;

        if ((logid >= NR_LOGS) || (logreg[logid].handle != req->rq_xprt)) {
            res = -1;
            return &res;
        }
        logreg[logid].handle = NULL;
        _closelog(logreg[logid].filp);
        res = 0;
        return &res;
    }

    /*
     * When there is a request to write a message to the log,
     * write_log_1_svc is called
     */
        void*
    writelog_1_svc(int logid, log_severity severity, char* message,
                  struct svc_req* req)
    {
        if ((logid >= NR_LOGS) || (logreg[logid].handle != req->rq_xprt)) {
            return NULL;
        }
      /*
       * Write message to file
       */
        fprintf(logreg[logid].filp, "%s (%s): %s\n",
                logreg[logid].ident, severityname[severity], message);
        return NULL;
    }

        static void
    close_handler(const SVCXPRT* handle, const bool_t dummy)
    {
        int i;
```

```
  /*
   * When the client dies, the log is closed with closelog
   */
    for (i = 0; i < NR_LOGS; ++i) {
        if (handle == logreg[i].handle) {
            logreg[i].handle = NULL;
            _closelog(logreg[i].filp);
        }
    }
}
```

The `logClient.c` file shows a client using the log server.

```
#include "log.h"
#include <stdio.h>

#define MSG_SIZE 128

    void
usage()
{
    puts("Usage: logClient <logserver_addr>");
    exit(2);
}

    void
runClient(CLIENT* clnt)
{
    char msg[MSG_SIZE];
    int logID;
    int* result;

  /*
   * client opens a log
   */
    result = openlog_1("client", clnt);
    if (NULL == result) {
        clnt_perror(clnt, "openlog");
        return;
    }
    logID = *result;
    if (-1 == logID) {
        puts("Cannot open the log.");
        return;
    }

    while(1) {
        struct rpc_err e;

    /*
      * Client writes a message in the log
```

```
     */
       puts("Enter a message in the log (\".\" to quit):");
       fgets(msg, MSG_SIZE, stdin);
       /*
    * Remove trailing CR
    */
       msg[strlen(msg)-1] = 0;

       if (!strcmp(msg, ".")) break;

       if (writelog_1(logID, LOG_INFO, msg, clnt) == NULL) {
           clnt_perror(clnt, "writelog");
           return;
       }
    }
   /*
    * Client closes the log
    */
    result = closelog_1(logID, clnt);
    if (NULL == result) {
       clnt_perror(clnt, "closelog");
       return;
    }
    logID = *result;
    if (-1 == logID) {
       puts("Cannot close the log.");
       return;
    }
}

    int
main(int argc, char* argv[])
{
    char* serv_addr;
    CLIENT* clnt;

    if (argc != 2) usage();

    serv_addr = argv[1];

    clnt = clnt_create(serv_addr, LOG, LOG_VERS1, "tcp");

    if (NULL == clnt) {
       clnt_pcreateerror("Cannot connect to log server");
       exit(1);
    }
    runClient(clnt);

    clnt_destroy(clnt);
}
```

User File Descriptor Callbacks

User file descriptor callbacks enable you to register file descriptors with callbacks, specifying one or more event types. Now you can use an RPC server to handle file descriptors that were not written for the Oracle Solaris RPC library.

With previous versions of the Oracle Solaris RPC library, you could use a server to receive both RPC calls and non-RPC file descriptors only if you wrote your own server loop, or used a separate thread to contact the socket API.

For user file descriptor callbacks, two new functions have been added to the Oracle Solaris RPC library, `svc_add_input(3NSL)` and `svc_remove_input(3NSL)`, to implement user file descriptor callbacks. These functions declare or remove a callback on a file descriptor.

When using this new callback feature you must:

- Create your `callback()` function by writing user code with the following syntax:

    ```
    typedef void (*svc_callback_t) (svc_input_id_t id, int fd, \
    unsigned int revents, void* cookie);
    ```

    The four parameters passed to the `callback()` function are:

`id`	Provides an identifier for each callback. This identifier can be used to remove a callback.
`fd`	The file descriptor that your callback is waiting for.
`revents`	An unsigned integer representing the events that have occurred. This set of events is a subset of the list given when the callback is registered.
`cookie`	The cookie given when the callback is registered. This cookie can be a pointer to specific data the server needs during the callback.

- Call `svc_add_input()` to register file descriptors and associated events, such as read or write, that the server must be aware of.

    ```
    svc_input_id_t svc_add_input (int fd, unsigned int revents, \
    svc_callback_t callback, void* cookie);
    ```

    A list of the events that can be specified is given in `poll(2)` .

- Specify a file descriptor. This file descriptor can be an entity such as a socket or a file.

When one of the specified events occurs, the standard server loop calls the user code through `svc_run()` and your callback performs the required operation on the file descriptor, socket or file.

When you no longer need a particular callback, call `svc_remove_input()` with the corresponding identifier to remove the callback.

Example of User File Descriptors

This example shows you how to register a user file descriptor on an RPC server and how to provide user defined callbacks. With this example you can monitor the time of day on both the server and the client.

The makefile for this example is shown below.

```
    RPCGEN = rpcgen

CLIENT = todClient
CLIENT_SRC = todClient.c timeofday_clnt.c
CLIENT_OBJ = $(CLIENT_SRC:.c=.o)

SERVER = todServer
SERVER_SRC = todServer.c timeofday_svc.c
SERVER_OBJ = $(SERVER_SRC:.c=.o)

RPCGEN_FILES = timeofday_clnt.c timeofday_svc.c  timeofday.h

CFLAGS += -I.

RPCGEN_FLAGS = -N -C
LIBS = -lsocket -lnsl

all: ./$(CLIENT) ./$(SERVER)


$(CLIENT): timeofday.h $(CLIENT_OBJ)
     cc -o $(CLIENT) $(LIBS) $(CLIENT_OBJ)

$(SERVER): timeofday.h $(SERVER_OBJ)
     cc -o $(SERVER)  $(LIBS) $(SERVER_OBJ)

timeofday_clnt.c: timeofday.x
      $(RPCGEN) -l $(RPCGEN_FLAGS) timeofday.x > timeofday_clnt.c

timeofday_svc.c: timeofday.x
     $(RPCGEN) -m $(RPCGEN_FLAGS) timeofday.x > timeofday_svc.c

timeofday.h: timeofday.x
     $(RPCGEN) -h $(RPCGEN_FLAGS) timeofday.x > timeofday.h


clean:
     rm -f $(CLIENT_OBJ) $(SERVER_OBJ) $(RPCGEN_FILES)
```

The `timeofday.x` file defines the RPC services offered by the server in this example. The services in this examples are `gettimeofday()` and `settimeofday()`.

```
program TIMEOFDAY {
    version VERS1 {
        int SENDTIMEOFDAY(string tod) = 1;

        string GETTIMEOFDAY() = 2;
    } = 1;
} = 0x20000090;
```

The `userfdServer.h` file defines the structure of messages sent on the sockets in this example.

```
   #include "timeofday.h"
#define PORT_NUMBER 1971

/*
 * Structure used to store data for a connection.
  * (user fds test).
 */
typedef struct {
 /*
 * Ids of the callback registration for this link.
 */
    svc_input_id_t in_id;
    svc_input_id_t out_id;

    /*
    * Data read from this connection.
    */
    char in_data[128];

    /*
    * Data to be written on this connection.
    */
    char out_data[128];
    char* out_ptr;

} Link;

    void
socket_read_callback(svc_input_id_t id, int fd, unsigned int events,
        void* cookie);
    void
socket_write_callback(svc_input_id_t id, int fd, unsigned int events,
                    void* cookie);
    void
socket_new_connection(svc_input_id_t id, int fd, unsigned int events,
                    void* cookie);

    void
timeofday_1(struct svc_req *rqstp, register SVCXPRT *transp);
```

The `todClient.c` file shows how the time of day is set on the client. In this file, RPC is used with and without sockets.

```
    #include "timeofday.h"

#include <stdio.h>
#include <netdb.h>
#define PORT_NUMBER 1971

    void
runClient();
    void
runSocketClient();


char* serv_addr;

    void
usage()
{
    puts("Usage: todClient [-socket] <server_addr>");
    exit(2);
}

    int
main(int argc, char* argv[])
{
    CLIENT* clnt;
    int sockClient;

    if ((argc != 2) && (argc != 3))
        usage();

    sockClient = (strcmp(argv[1], "-socket") == 0);

  /*
   * Choose to use sockets (sockClient).
   * If sockets are not available,
   * use RPC without sockets (runClient).
   */
    if (sockClient && (argc != 3))
        usage();

    serv_addr = argv[sockClient? 2:1];

    if (sockClient) {
        runSocketClient();
    } else {
        runClient();
    }
```

```
    return 0;
}
/*
 * Use RPC without sockets
 */
    void
runClient()
{
    CLIENT* clnt;
    char* pts;
    char** serverTime;

    time_t now;

    clnt = clnt_create(serv_addr, TIMEOFDAY, VERS1, "tcp");
    if (NULL == clnt) {
        clnt_pcreateerror("Cannot connect to log server");
        exit(1);
    }

    time(&now);
    pts = ctime(&now);

    printf("Send local time to server\n");

    /*
    * Set the local time and send this time to the server.
    */
    sendtimeofday_1(pts, clnt);

    /*
    * Ask the server for the current time.
    */
    serverTime = gettimeofday_1(clnt);

    printf("Time received from server: %s\n", *serverTime);

    clnt_destroy(clnt);
}

/*
 * Use RPC with sockets
 */
    void
runSocketClient()
/*
 * Create a socket
 */
{
    int s = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in sin;
```

```
char* pts;
char buffer[80];
int len;

time_t now;
struct hostent* hent;
unsigned long serverAddr;

if (-1 == s) {
    perror("cannot allocate socket.");
    return;
}

hent = gethostbyname(serv_addr);
if (NULL == hent) {
    if ((int)(serverAddr = inet_addr(serv_addr)) == -1) {
        puts("Bad server address");
        return;
    }
} else {
    memcpy(&serverAddr, hent->h_addr_list[0], sizeof(serverAddr));
}

sin.sin_port = htons(PORT_NUMBER);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = serverAddr;

/*
 * Connect the socket
 */
if (-1 == connect(s, (struct sockaddr*)(&sin),
                  sizeof(struct sockaddr_in))) {
    perror("cannot connect the socket.");
    return;
}

time(&now);
pts = ctime(&now);

/*
 * Write a message on the socket.
 * The message is the current time of the client.
 */
puts("Send the local time to the server.");
if (-1 == write(s, pts, strlen(pts)+1)) {
    perror("Cannot write the socket");
    return;
}

/*
 * Read the message on the socket.
```

```
    * The message is the current time of the server
    */
    puts("Get the local time from the server.");
    len = read(s, buffer, sizeof(buffer));

    if (len == -1) {
        perror("Cannot read the socket");
        return;
    }
    puts(buffer);

    puts("Close the socket.");
    close(s);
}
```

The todServer.c file shows the use of the timeofday service from the server side.

```
   #include "timeofday.h"
#include "userfdServer.h"
#include <stdio.h>
#include <errno.h>
#define PORT_NUMBER 1971

int listenSocket;

/*
 * Implementation of the RPC server.
 */

    int*
sendtimeofday_1_svc(char* time, struct svc_req* req)
{
    static int result = 0;

    printf("Server: Receive local time from client  %s\n", time);
    return &result;
}

    char **
gettimeofday_1_svc(struct svc_req* req)
{
    static char buff[80];
    char* pts;
    time_t now;
    static char* result = &(buff[0]);

    time(&now);
    strcpy(result, ctime(&now));

    return &result;
}
```

```
/*
 * Implementation of the socket server.
 */

    int
create_connection_socket()
{
    struct sockaddr_in sin;
    int size = sizeof(struct sockaddr_in);
    unsigned int port;

  /*
   * Create a socket
   */
   listenSocket = socket(PF_INET, SOCK_STREAM, 0);

   if (-1 == listenSocket) {
       perror("cannot allocate socket.");
       return -1;
   }

   sin.sin_port = htons(PORT_NUMBER);
   sin.sin_family = AF_INET;
   sin.sin_addr.s_addr = INADDR_ANY;

   if (bind(listenSocket, (struct sockaddr*)&sin, sizeof(sin)) == -1) {
       perror("cannot bind the socket.");
       close(listenSocket);
       return -1;
   }
  /*
   * The server waits for the client
   * connection to be created
   */
   if (listen(listenSocket, 1)) {
       perror("cannot listen.");
       close(listenSocket);
       listenSocket = -1;
       return -1;
   }

  /*
   * svc_add_input registers a read callback,
   * socket_new_connection, on the listening socket.
   * This callback is invoked when a new connection
   * is pending. */
   if (svc_add_input(listenSocket, POLLIN,
                     socket_new_connection, (void*) NULL) == -1) {
       puts("Cannot register callback");
       close(listenSocket);
       listenSocket = -1;
```

```
        return -1;
    }

    return 0;
}

/*
 * Define the socket_new_connection callback function
 */
    void
socket_new_connection(svc_input_id_t id, int fd,
                        unsigned int events, void* cookie)
{
    Link* lnk;
    int connSocket;

  /*
    * The server is called when a connection is
    * pending on the socket. Accept this connection now.
    * The call is non-blocking.
    * Create a socket to treat the call.
    */
    connSocket = accept(listenSocket, NULL, NULL);
    if (-1 == connSocket) {
        perror("Server: Error: Cannot accept a connection.");
        return;
    }

    lnk = (Link*)malloc(sizeof(Link));
    lnk->in_data[0] = 0;

  /*
    * New callback created, socket_read_callback.
    */
    lnk->in_id = svc_add_input(connSocket, POLLIN,
          socket_read_callback, (void*)lnk);
}
/*
 * New callback, socket_read_callback, is defined
 */
    void
socket_read_callback(svc_input_id_t id, int fd, unsigned int events,
                      void* cookie)
{
    char buffer[128];
    int len;
    Link* lnk = (Link*)cookie;

    /*
     * Read the message. This read call does not block.
     */
```

```
            len = read(fd, buffer, sizeof(buffer));

            if (len > 0) {
               /*
            * Got some data. Copy it in the buffer
            * associated with this socket connection.
            */
                strncat (lnk->in_data, buffer, len);

                /*
            * Test if we receive the complete data.
                 * Otherwise, this is only a partial read.
            */
                if (buffer[len-1] == 0) {
                    char* pts;
                    time_t now;

                    /*
             * Print the time of day you received.
             */
                    printf("Server: Got time of day from the client: \n %s",
                            lnk->in_data);

                    /*
             * Setup the reply data
             * (server current time of day).
             */
                    time(&now);
                    pts = ctime(&now);

                    strcpy(lnk->out_data, pts);
                    lnk->out_ptr = &(lnk->out_data[0]);

             /*
             * Register a write callback (socket_write_callback)
             * that does not block when writing a reply.
             * You can use POLLOUT when you have write
             * access to the socket
             */
                    lnk->out_id = svc_add_input(fd, POLLOUT,
                                            socket_write_callback, (void*)lnk);

                }
            } else if (len == 0) {
            /*
             * Socket closed in peer.  Closing the socket.
             */
        close(fd);
            } else {
                /*
            * Has the socket been closed by peer?
```

```
    */
        if (errno != ECONNRESET) {
            /*
     * If no, this is an error.
     */
            perror("Server: error in reading the socket");
            printf("%d\n", errno);
        }
        close(fd);
    }
}

/*
 * Define the socket_write_callback.
 * This callback is called when you have write
 * access to the socket.
 */
    void
socket_write_callback(svc_input_id_t id, int fd, unsigned int events,
                      void* cookie)
{
    Link* lnk = (Link*)cookie;

    /*
    * Compute the length of remaining data to write.
    */
    int len = strlen(lnk->out_ptr)+1;

/*
 * Send the time to the client
 */
    if (write(fd, lnk->out_ptr, len) == len) {
    /*
    * All data sent.
    */

     /*
      * Unregister the two callbacks. This unregistration
       * is demonstrated here as the registration is
      * removed automatically when the file descriptor
    * is closed.
    */
        svc_remove_input(lnk->in_id);
        svc_remove_input(lnk->out_id);


        /*
    * Close the socket.
    */
        close(fd);
    }
}
```

```
     void
main()
{
    int res;

  /*
   * Create the timeofday service and a socket
   */
   res = create_connection_socket();
   if (-1 == res) {
       puts("server: unable to create the connection socket.\n");
       exit(-1);
   }

   res = svc_create(timeofday_1, TIMEOFDAY, VERS1, "tcp");
   if (-1 == res) {
       puts("server: unable to create RPC service.\n");
       exit(-1);
   }

  /*
   Poll the user file descriptors.
   */
   svc_run();
}
```

A

XDR Technical Note

This appendix is a technical note on the Oracle implementation of the external data representation (XDR) standard, which is a set of library routines that enable C programmers to describe arbitrary data structures in a system-independent manner.

What Is XDR?

XDR is the backbone of the Oracle Remote Procedure Call package. Data for RPCs are transmitted using this standard. XDR library routines should be used to transmit data accessed (read or written) by more than one type of system.

XDR works across different languages, operating systems, and system architectures. Most users (particularly RPC users) only need the information in the sections on number filters, floating point filters, and enumeration filters. Programmers who want to implement RPC and XDR on new systems should read this technical note and the protocol specification.

You can use `rpcgen` to write XDR routines even in cases where no RPC calls are being made.

C programs that use XDR routines must include the file `<rpc/xdr.h>`, which contains all the necessary interfaces to the XDR system. Because the library `libnsl.a` contains all the XDR routines, you compile it by typing:

```
example% cc program.c
```

In many environments, several criteria must be observed to accomplish porting. The ramifications of a small programmatic change are not always apparent, but they can often have far-reaching implications. Consider the program to read/write a line of text that is shown in the next two code examples.

EXAMPLE 76   Writer Example (initial)

```
#include <stdio.h>

main()          /* writer.c */
{  int i;
```

```
for (i = 0; i < 8; i++) {
  if (fwrite((char *) &i, sizeof(i), 1, stdout) != 1) {
   fprintf(stderr, "failed!\n");
   exit(1);
  }
 }
 exit(0);
}
```

EXAMPLE 77   Reader Example (initial)

```
#include <stdio.h>

main()       /* reader.c */
{
  int i, j;

 for (j = 0; j < 8; j++) {
   if (fread((char *) &i, sizeof(i), 1, stdin) != 1) {
    fprintf(stderr, "failed!\n");
    exit(1);
   }
   printf("%ld ", i);
  }
  printf("\n");
  exit(0);
}
```

The two programs appear to be portable because they:

- Pass lint checking
- Exhibit the same behavior when executed locally on any hardware architecture

Piping the output of the writer program to the reader program gives identical results on a SPARC or Intel system.

```
% writer | reader
0 1 2 3 4 5 6 7
%
intel% writer | reader
0 1 2 3 4 5 6 7
intel%
```

With the advent of local area networks and 4.2BSD came the concept of "network pipes," which is a process that produces data on one system and a second process that consumes data on another system. You can construct a network pipe with writer and reader. Here are the results if the writer produces data on a SPARC system, and the reader consumes data on Intel architecture.

```
% writer | rsh intel reader
0 16777216 33554432 50331648 67108864 83886080 100663296
```

```
117440512
%
```

Executing `writer` on the Intel and `reader` on the SPARC system produces identical results. These results occur because the byte ordering of data differs between the Intel and the SPARC, even though the word size is the same.

Note - 16777216 is 2^{24}. When 4 bytes are reversed, the 1 is placed in the 24th bit.

Whenever data is shared by two or more system types, a need exists for portable data. You can make data portable by replacing the `read()` and `write()` calls with calls to an XDR library routine, `xdr_int()`, a filter that knows the standard representation of an int integer in its external form. The revised versions of `writer` are shown in the following code example.

EXAMPLE 78 Writer Example (XDR modified)

```
#include <stdio.h>

#include <rpc/rpc.h> /* xdr is a sub-library of rpc */

main()        /* writer.c */
{
  XDR xdrs;
  int i;

xdrstdio_create(&xdrs, stdout, XDR_ENCODE);

  for (i = 0; i < 8; i++) {
   if (!xdr_int(&xdrs, &i)) {
    fprintf(stderr, "failed!\n");
    exit(1);
   }
  }
  exit(0);
}
```

The following code example shows the revised versions of `reader`.

EXAMPLE 79 Reader Example (XDR modified)

```
#include <stdio.h>

#include <rpc/rpc.h> /* xdr is a sub-library of rpc */

main()                /* reader.c */
{
  XDR xdrs;
  int i, j;
```

```
xdrstdio_create(&xdrs, stdin, XDR_DECODE);

  for (j = 0; j < 8; j++) {
   if (!xdr_int(&xdrs, &i)) {
    fprintf(stderr, "failed!\n");
    exit(1);
   }
   printf("%ld ", i);
  }
  printf("\n");
  exit(0);
}
```

The new programs were executed on a SPARC system, on an Intel, and from a SPARC to an Intel. The results follow.

```
% writer | reader
0 1 2 3 4 5 6 7
%
intel% writer | reader
0 1 2 3 4 5 6 7
intel%
% writer | rsh intel reader
0 1 2 3 4 5 6 7


%
```

Note - Arbitrary data structures can create portability issues, particularly with respect to alignment and pointers. Alignment on word boundaries cause the size of a structure to vary from system to system. Pointers, which are very convenient to use, have no meaning outside the system where they are defined.

Canonical Standard

The XDR approach to standardizing data representations is canonical. That is, XDR defines a single byte order, a single floating-point representation (IEEE), and so on. Any program running on any system can use XDR to create portable data by translating its local representation to the XDR standard representations. Similarly, any program running on any system can read portable data by translating the XDR standard representations to its local equivalents.

The single standard completely decouples programs that create or send portable data from those that use or receive portable data. A new system learns how to convert the standard representations and its local representations. The local representations of other systems are irrelevant. Conversely, the local representations of the new system are also irrelevant to

existing programs running on other systems. Such programs can immediately read portable data produced by the new system because such data conforms to the canonical standards that they already understand.

Strong precedents are in place for XDR's canonical approach. For example, TCP/IP, UDP/IP, XNS, Ethernet, and, indeed, all protocols below layer five of the ISO model, are canonical protocols. The advantage of any canonical approach is simplicity. In the case of XDR, a single set of conversion routines is written once and is never touched again.

The canonical approach has the disadvantage of unnecessary conversion to and from the XDR standard when data is transferred between two systems with identical byte order. Suppose two Intel systems are transferring integers according to the XDR standard. The sending system converts the integers from Intel host byte order to XDR byte order. The receiving system performs the reverse conversion. Because both systems observe the same byte order, their conversions are unnecessary.

The time spent converting to and from a canonical representation is insignificant, especially in distributed applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure.

To transmit a tree, for example, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there. Storage for the leaf might have to be de-allocated as well. Similarly, to receive a tree, you must allocate storage for each leaf, move data from the buffer to the leaf and properly align it, and construct pointers to link the leaves together.

Every system pays the cost of traversing and copying data structures whether or not conversion is required. In distributed applications, communications overhead, which is the time required to move the data down the sender's protocol layers, across the network, and up the receiver's protocol layers, dwarfs conversion overhead.

XDR Library

The XDR library solves data portability problems, and also enables you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, using the library can be helpful even when the data is not shared among systems on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves might contain arrays of arbitrary elements, or pointers to other structures.

Look closely at Example 78, "Writer Example (XDR modified)," on page 177 and Example 79, "Reader Example (XDR modified)," on page 177. A family of XDR stream-creation routines has each member treat the stream of bits differently. In the example, data is manipulated using standard I/O routines, so you use `xdrstdio_create()`. The

parameters to XDR stream-creation routines vary according to their function. In the example, `xdrstdio_create()` takes a pointer to an XDR structure that it initializes, a pointer to a `FILE` that the input or output is performed on, and the operation. The operation might be `XDR_ENCODE` for serializing in the writer program, or `XDR_DECODE` for deserializing in the reader program.

Note - RPC users never need to create XDR streams. The RPC system itself creates these streams, which are then passed to the users.

The `xdr_int()` primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns `FALSE` (0) if it fails, and `TRUE` (1) if it succeeds. Second, for each data type, `xxx`, there is an associated XDR routine of the form:

```
xdr_xxx(xdrs, xp)
    XDR *xdrs;
    xxx *xp;
{
}
```

In this case, `xxx` is int, and the corresponding XDR routine is a primitive, `xdr_int()`. The client could also define an arbitrary structure `xxx`, in which case the client would also supply the routine `xdr_xxx()`, describing each field by calling XDR routines of the appropriate type. In all cases the first parameter, *xdrs*, can be treated as an opaque handle and passed to the primitive routines.

An XDR routine is direction independent. That is, the same routine is called to serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation. This practice almost guarantees that serialized data can also be deserialized.

This one routine is used by both producer and consumer of networked data. This concept is implemented by always passing the address of an object rather than the object itself. Only in the case of deserialization is the object modified. This feature is not shown in the example, but its value becomes obvious when nontrivial data structures are passed among systems. If needed, the user can obtain the direction of the XDR operation. For details, see the section"Operation Directions" on page 195.

A slightly more complicated example follows. Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type.

```
struct gnumbers {
    int g_assets;
    int g_liabilities;
};
```

The corresponding XDR routine describing this structure is as follows.

```
bool_t                    /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
```

```
    XDR *xdrs;
    struct gnumbers *gp;
{
    if (xdr_int(xdrs, &gp->g_assets) &&
          xdr_int(xdrs, &gp->g_liabilities))
       return(TRUE);
    return(FALSE);
}
```

Note that the parameter *xdrs* is never inspected or modified. It is only passed on to the subcomponent routines. The routine must inspect the return value of each XDR routine call, and to halt immediately and return FALSE if the subroutine fails.

This example also shows that the type bool_t is declared as an integer that has only the values TRUE (1) and FALSE (0). This document uses the following definitions:

```
#define bool_t int
#define TRUE 1
#define FALSE 0
```

By observing these conventions, you can rewrite xdr_gnumbers() as follows:

```
xdr_gnumbers(xdrs, gp)
    XDR *xdrs;
    struct gnumbers *gp;
{
    return(xdr_int(xdrs, &gp->g_assets) &&
            xdr_int(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

XDR Library Primitives

This section provides a synopsis of each XDR primitive. It starts with memory allocation and the basic data types, then moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file <rpc/xdr.h>, automatically included by <rpc/rpc.h>.

Memory Requirements for XDR Routines

When using XDR routines, you sometimes need to to pre-allocate memory, or to determine memory requirements. When you need to control the allocation and de-allocation of memory for XDR conversion routines, you can use a routine, xdr_sizeof(). This routine returns the number of bytes needed to encode and decode data with one of the XDR filter functions (func()). The output of the xdr_sizeof() function does not include RPC headers or

record markers. You must add them to get a complete accounting of the memory required. `xdr_sizeof()` returns a zero on error.

```
xdr_sizeof(xdrproc_t func, void *data)
```

Use `xdr_sizeof()` for the allocation of memory in applications that use XDR outside of the RPC environment, to select between transport protocols, and at the lower levels of RPC to perform client and server creation functions.

The next two code examples illustrate two uses of `xdr_sizeof()`.

EXAMPLE 80   `xdr_sizeof` Example #1

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

/*
 * This function takes as input a CLIENT handle, an XDR function
and
 * a pointer to data to be XDR'd. It returns TRUE if the amount of
 * data to be XDR'd may be sent using the transport associated
with
 * the CLIENT handle, and false otherwise.
 */
bool_t
cansend(cl, xdrfunc, xdrdata)
 CLIENT *cl;
 xdrproc_t xdrfunc;
 void   *xdrdata;
{
 int fd;
 struct t_info tinfo;

 if (clnt_control(cl, CLGET_FD, &fd) == -1) {
  /* handle clnt_control() error */
  return (FALSE);
 }

 if (t_getinfo(fd, &tinfo) == -1) {
  /* handle t_getinfo() error */
  return (FALSE);
 } else {
  if (tinfo.servtype == T_CLTS) {
   /*
    * This is a connectionless transport. Use xdr_sizeof()
    * to compute the size of this request to see whether it
    * is too large for this transport.
    */
   switch(tinfo.tsdu) {
    case 0:                    /* no concept of TSDUs */
    case -2:                    /* can't send normal data */
```

```
    return (FALSE);
    break;
   case -1:                        /* no limit on TSDU size */
    return (TRUE);
    break;
   default:
    if (tinfo.tsdu < xdr_sizeof(xdrfunc, xdrdata))
     return (FALSE);
    else
     return (TRUE);
  }
 } else
  return (TRUE);
 }
}
```

The second xdr_sizeof() example follows.

EXAMPLE 81   xdr_sizeof Example #2

```
#include <sys/statvfs.h>

#include <sys/sysmacros.h>

/*
 * This function takes as input a file name, an XDR function, and
a
 * pointer to data to be XDR'd. It returns TRUE if the filesystem
 * on which the file resides has room for the additional amount
of
 * data to be XDR'd. Note that since the information statvfs(2)
 * returns about the filesystem is in blocks you must convert the
 * value returned by xdr_sizeof() from bytes to disk blocks.
 */
bool_t
canwrite(file, xdrfunc, xdrdata)
 char     *file;
 xdrproc_t xdrfunc;
 void     *xdrdata;
{
 struct statvfs s;

 if (statvfs(file, &s) == -1) {
  /* handle statvfs() error */
  return (FALSE);
 }

 if (s.f_bavail >= btod(xdr_sizeof(xdrfunc, xdrdata)))
  return (TRUE);
 else
```

```
   return (FALSE);
}
```

Number Filters

The XDR library provides primitives to translate between numbers and their corresponding external representations. Primitives cover the set of numbers in the types:

```
[signed, unsigned] * [short, int, long]
```

Specifically, the eight primitives are:

```
bool_t xdr_char(xdrs, op)
   XDR *xdrs;
   char *cp;
bool_t xdr_u_char(xdrs, ucp)
   XDR *xdrs;
   unsigned char *ucp;
bool_t xdr_int(xdrs, ip)
   XDR *xdrs;
   int *ip;
bool_t xdr_u_int(xdrs, up)
   XDR *xdrs;
   unsigned *up;
bool_t xdr_long(xdrs, lip)
   XDR *xdrs;
   long *lip;
bool_t xdr_u_long(xdrs, lup)
   XDR *xdrs;
   u_long *lup;
bool_t xdr_short(xdrs, sip)
   XDR *xdrs;
   short *sip;
bool_t xdr_u_short(xdrs, sup)
   XDR *xdrs;
   u_short *sup;
```

The first parameter, *xdrs*, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

Floating-Point Filters

The XDR library also provides primitive routines for C floating-point types.

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

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

The first parameter, *xdrs*, is an XDR stream handle. The second parameter is the address of the floating-point number that provides data to the stream or receives data from it. Both routines return TRUE if they complete successfully, and FALSE otherwise.

Note - Because the numbers are represented in IEEE floating point, routines might fail when decoding a valid IEEE representation into a system-specific representation, or the reverse.

Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C enum has the same representation inside the system as a C integer. The Boolean type is an important instance of the enum. The external representation of a Boolean is always TRUE (1) or FALSE (0).

```
#define bool_t int
#define FALSE  0
#define TRUE   1
#define enum_t int
bool_t xdr_enum(xdrs, ep)
   XDR *xdrs;
   enum_t *ep;
bool_t xdr_bool(xdrs, bp)
   XDR *xdrs;
   bool_t *bp;
```

The second parameters *ep* and *bp* are addresses of the associated type that provides data to or receives data from the stream xdrs.

No-Data Routine

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine.

```
bool_t xdr_void(); /* always returns TRUE */
```

Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives discussed previously. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives can use memory management. In many cases, memory is allocated when deserializing data with `XDR_DECODE`. Therefore, the `XDR` package must provide a means to de-allocate memory. The `XDR` operation, `XDR_FREE` provides this means. To review, the three `XDR` directional operations are `XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`.

Strings

In the C language, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a `char *`, and not a sequence of characters. The external representation of a string is drastically different from its internal representation.

Externally, strings are represented as sequences of ASCII characters. Internally, strings are represented with character pointers. Conversion between the two representations is accomplished with the routine `xdr_string()`:

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

The first parameter *xdrs* is the XDR stream handle. The second parameter *sp* is a pointer to a string (type `char **`). The third parameter *maxlength* specifies the maximum number of bytes allowed during encoding or decoding. Its value is usually specified by a protocol. For example, a protocol specification might say that a file name can be no longer than 255 characters. The routine returns `FALSE` if the number of characters exceeds *maxlength*, and `TRUE` if it doesn't.

The behavior of `xdr_string()` is similar to the behavior of other routines discussed in this section. For example, in the direction `XDR_ENCODE`, the parameter *sp* points to a string of a certain length. If the string does not exceed *maxlength*, the bytes are serialized.

The effect of deserializing a string is subtle. First the length of the incoming string is determined; it must not exceed *maxlength*. Next *sp* is dereferenced. If the value is `NULL`, a string of the appropriate length is allocated and **sp* is set to this string. If the original value of **sp* is nonnull, the XDR package assumes that a target area has been allocated that can hold strings no longer than *maxlength*. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the `XDR_FREE` operation the string is obtained by dereferencing *sp*. If the string is not `NULL`, it is freed and **sp* is set to `NULL`. In this operation `xdr_string()` ignores the *maxlength* parameter.

Note that you can use XDR on an empty string ("") but not on a NULL string.

Byte Arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways:

- The length of the array, the byte count, is explicitly located in an unsigned integer.
- The byte sequence is not terminated by a null character.
- The external representation of the bytes is the same as their internal representation.

The primitive `xdr_bytes()` converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
    XDR *xdrs;
    char **bpp;
    u_int *lp;
    u_int maxlength;
```

The usage of the first, second, and fourth parameters is identical to the first, second, and third parameters of `xdr_string()`. The length of the byte area is obtained by dereferencing *lp* when serializing; **lp* is set to the byte length when deserializing.

Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. The `xdr_bytes()` routine treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is built in. The generic array primitive, `xdr_array()` requires parameters identical to those of `xdr_bytes()` plus two more: the size of array elements, and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsize, xdr_element)
    XDR *xdrs;
    char **ap;
    u_int *lp;
    u_int maxlength;
    u_int elementsize;
    bool_t (*xdr_element)();
```

The parameter *ap* is the address of the pointer to the array. If **ap* is NULL when the array is being deserialized, XDR allocates an array of the appropriate size and sets **ap* to that array. The element count of the array is obtained from **lp* when the array is serialized; **lp* is set to the array length when the array is deserialized. The parameter *maxlength* is the maximum number of elements that the array is allowed to have; *elementsiz* is the byte size of each element of the

array (the C function `sizeof()` can be used to obtain this value). The `xdr_element()` routine is called to serialize, deserialize, or free each element of the array.

Before defining more constructed data types, three examples are presented.

Array Example 1

A user on a networked system can be identified by

- The system name.
- The user's UID. See the getuid(2) man page.
- The group numbers to which the user belongs. See the getgroups(2) man page.

A structure with this information and its associated XDR routine could be coded as in the following code example.

EXAMPLE 82   Array Example #1

```
struct netuser {
  char  *nu_systemname;
  int   nu_uid;
  u_int nu_glen;
  int   *nu_gids;
 };
#define NLEN 255       /* system names < 256 chars */
#define NGRPS 20       /* user can't be in > 20 groups */

bool_t
xdr_netuser(xdrs, nup)
  XDR *xdrs;
  struct netuser *nup;
{
  return(xdr_string(xdrs, &nup->nu_systemname, NLEN) &&
       xdr_int(xdrs, &nup->nu_uid) &&
       xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen, NGRPS,
                 sizeof (int), xdr_int));
}
```

Array Example 2

You could implement a party of network users as an array of `netuser` structure. The declaration and its associated XDR routines are as shown in the following code example.

EXAMPLE 83   Array Example #2

```
struct party {
```

```
  u_int p_len;
  struct netuser *p_nusers;
};
#define PLEN 500 /* max number of users in a party */
bool_t
xdr_party(xdrs, pp)
  XDR *xdrs;
  struct party *pp;
{
  return(xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
   sizeof (struct netuser), xdr_netuser));
}
```

Array Example 3

You can combine the well-known parameters to main, *argc* and *argv*, into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like the following example.

EXAMPLE 84   Array Example #3

```
struct cmd {
  u_int c_argc;
  char **c_argv;
};
#define ALEN 1000          /* args cannot be > 1000 chars */
 #define NARGC 100          /* commands cannot have > 100 args */

struct history {
  u_int h_len;
  struct cmd *h_cmds;
};
#define NCMDS 75           /* history is no more than 75 commands */

bool_t
xdr_wrapstring(xdrs, sp)

  XDR *xdrs;
  char **sp;
{
  return(xdr_string(xdrs, sp, ALEN));

}

bool_t
xdr_cmd(xdrs, cp)

  XDR *xdrs;
```

```
  struct cmd *cp;
{
  return(xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
          sizeof (char *), xdr_wrapstring));
}
bool_t
xdr_history(xdrs, hp)

  XDR *xdrs;
  struct history *hp;
{
  return(xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
          sizeof (struct cmd), xdr_cmd));
}
```

Some confusion in this example is that you need the routine `xdr_wrapstring()` to package the `xdr_string()` routine, because the implementation of `xdr_array()` passes only two parameters to the array element description routine. `xdr_wrapstring()` supplies the third parameter to `xdr_string()`.

By now, the recursive nature of the XDR library should be obvious. A discussion follows of more constructed data types.

Opaque Data

In some protocols, handles are passed from a server to the client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is, handles are opaque. The `xdr_opaque()` primitive is used for describing fixed-sized opaque bytes.

```
bool_t
xdr_opaque(xdrs, p, len)
   XDR *xdrs;
   char *p;
   u_int len;
```

The parameter *p* is the location of the bytes, *len* is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object is not system portable.

There is another routine i n the Oracle Solaris system for manipulating opaque data. This routine, the `xdr_netobj`, sends counted opaque data, much like `xdr_opaque()`. The following code example illustrates the syntax of `xdr_netobj()`.

EXAMPLE 85   xdr_netobj Routine

```
struct netobj {
```

```
 u_int   n_len;
 char    *n_bytes;
};
typedef struct netobj netobj;

bool_t
xdr_netobj(xdrs, np)
 XDR *xdrs;
 struct netobj *np;
```

The `xdr_netobj()` routine is a filter primitive that translates between variable-length opaque data and its external representation. The parameter np is the address of the netobj structure containing both a length and a pointer to the opaque data. The length may be no more than MAX_NETOBJ_SZ bytes. This routine returns TRUE if it succeeds, FALSE otherwise.

Fixed-Length Arrays

The XDR library provides a primitive, `xdr_vector()`, for fixed-length arrays, shown in the following code example.

EXAMPLE 86   `xdr_vector` Routine

```
#define NLEN 255 /* system names must be < 256 chars */
#define NGRPS 20 /* user belongs to exactly 20 groups */

struct netuser {
  char *nu_systemname;
  int nu_uid;
  int nu_gids[NGRPS];
};

bool_t
xdr_netuser(xdrs, nup)
  XDR *xdrs;
  struct netuser *nup;
{
  int i;

 if (!xdr_string(xdrs, &nup->nu_systemname, NLEN))
   return(FALSE);
  if (!xdr_int(xdrs, &nup->nu_uid))
   return(FALSE);
  if (!xdr_vector(xdrs, nup->nu_gids, NGRPS, sizeof(int),
      xdr_int))
   return(FALSE);
  return(TRUE);
}
```

Discriminated Unions

The XDR library supports discriminated unions. A discriminated union is a C union and an
`enum_t` value that selects an "arm" of the union.

```
struct xdr_discrim {
   enum_t value;
   bool_t (*proc)();
};

bool_t
 xdr_union(xdrs, dscmp, unp, arms, defaultarm)
   XDR *xdrs;
   enum_t *dscmp;
   char *unp;
   struct xdr_discrim *arms;
   bool_t (*defaultarm)(); /* may equal NULL */
```

First the routine translates the discriminant of the union located at *dscmp*. The discriminant is
always an `enum_t`. Next the union located at *unp* is translated. The parameter *arms* is a pointer
to an array of `xdr_discrim` structures. Each structure contains an ordered pair of *[value,proc]*.
If the union's discriminant is equal to the associated *value*, then the `proc` is called to translate
the union. The end of the `xdr_discrim` structure array is denoted by a routine of value `NULL` (0).
If the discriminant is not found in the *arms* array, then the `defaultarm()` procedure is called if
it is nonnull. Otherwise the routine returns `FALSE`.

Discriminated Union Example

Suppose the type of a union is integer, character pointer (a string), or a `gnumbers` structure.
Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype {INTEGER=1, STRING=2, GNUMBERS=3};
struct u_tag {
   enum utype utype; /* the union's discriminant */
   union {
      int ival;
      char *pval;
      struct gnumbers gn;
   } uval;
};
```

The following code example constructs an XDR procedure to deserialize the discriminated
union.

EXAMPLE 87   XDR Discriminated Union

```
struct xdr_discrim u_tag_arms[4] = {
```

```
  {INTEGER, xdr_int},
  {GNUMBERS, xdr_gnumbers}
  {STRING, xdr_wrapstring},
  {__dontcare__, NULL}
  /* always terminate arms with a NULL xdr_proc */
 }

bool_t
xdr_u_tag(xdrs, utp)
  XDR *xdrs;
  struct u_tag *utp;
{
  return(xdr_union(xdrs, &utp->utype, &utp->uval,
       u_tag_arms, NULL));
}
```

The routine `xdr_gnumbers()` was presented previously in "XDR Library" on page 179. The default *arm* parameter to `xdr_union()`, the last parameter, is NULL in this example. Therefore, the value of the union's discriminant can legally take on only values listed in the *u_tag_arms* array. Example 87, "XDR Discriminated Union," on page 192 also demonstrates that the elements of the arm's array do not need to be sorted.

The values of the discriminant can be sparse, though in Example 87, "XDR Discriminated Union," on page 192 they are not. Make a practice of assigning explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Pointers

In C, putting pointers to another structure within a structure is often convenient. The `xdr_reference()` primitive makes it easy to serialize, deserialize, and free these referenced structures.

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

Parameter *pp* is the address of the pointer to the structure; parameter *ssize* is the size in bytes of the structure (use the C function `sizeof()` to obtain this value); and `proc()` is the XDR routine that describes the structure. When decoding data, storage is allocated if **pp* is NULL.

A primitive `xdr_struct()` does not need to describe structures within structures because pointers are always sufficient.

Pointer Example

Suppose you have a structure containing a person's name and a pointer to a `gnumbers` structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
    char *name;
    struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
    XDR *xdrs;
    struct pgn *pp;
{
    return(xdr_string(xdrs, &pp->name, NLEN) &&
        xdr_reference(xdrs, &pp->gnp, sizeof(struct gnumbers),
                    xdr_gnumbers));
}
```

Pointer Semantics

In many applications, C programmers attach double meaning to the values of a pointer. Typically the value `NULL` (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For example, a `NULL` pointer value for *gnp* could indicate that the person's assets and liabilities are unknown. That is, the pointer value encodes two things: whether the data is known and, if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive `xdr_reference()` cannot and does not attach any special meaning to a null-value pointer during serialization. That is, passing an address of a pointer that has a value of value of `NULL` to `xdr_reference()` when serializing data most likely causes a memory fault and, on the UNIX system, a core dump.

`xdr_pointer()` correctly handles `NULL` pointers.

Nonfilter Primitives

You can manipulate XDR streams with the primitives discussed in this section.

```
u_int xdr_getpos(xdrs)
```

```
    XDR *xdrs;

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

xdr_destroy(xdrs)
    XDR *xdrs;
```

The routine `xdr_getpo()s()` returns an unsigned integer that describes the current position in the data stream.

> ⚠️ **Caution -** In some XDR streams, the value returned by `x()dr_getpos()` is meaningless; the routine returns a -1 in this case (though -1 should be a legitimate value).

The routine `xdr_setpos()` sets a stream position to *pos*. In some XDR streams, setting a position is impossible; in such cases, `xdr_setpos()` returns `FALSE`. This routine also fails if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The `xdr_destroy()` primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

Operation Directions

At times, you might want to optimize XDR routines by taking advantage of the direction of the operation: XDR_ENCODE, XDR_DECODE or XDR_FREE. The value `xdrs->x_op` always contains the direction of the XDR operation. An example in "Linked Lists" on page 200 demonstrates the usefulness of the `xdrs->x_op` field.

Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream. Streams currently exist for deserialization of data to or from standard I/O `FILE` streams, record streams, memory, and UNIX files.

Standard I/O Streams

XDR streams can be interfaced to standard I/O using the `xdrstdio_create()` routine.

```
#include <stdio.h>
#include <rpc/rpc.h> /* xdr is part of rpc */
```

```
void
xdrstdio_create(xdrs, fp, xdr_op)
   XDR *xdrs;
   FILE *fp;
   enum xdr_op x_op;
```

The routine `xdrstdio_create()` initializes an XDR stream pointed to by *xdrs*. The XDR stream interfaces to the standard I/O library. Parameter *fp* is an open file, and *x_op* is an XDR direction.

Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory:

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

void
xdrmem_create(xdrs, addr, len, x_op)
   XDR *xdrs;
   char *addr;
   u_int len;
   enum xdr_op x_op;
```

The routine `xdrmem_create()` initializes an XDR stream in local memory. The memory is pointed to by parameter *addr*. Parameter *len* is the length in bytes of the memory. The parameters *xdrs* and *x_op* are identical to the corresponding parameters of `xdrstdio_create()`. Currently, the datagram implementation of RPC uses `xdrmem_create()`. Complete call or result messages are built in memory before calling the `t_sndndata()` TLI routine.

Record TCP/IP Streams

A record stream is an XDR stream built on top of a record-marking standard that is built on top of the UNIX file or 4.2 BSD connection interface.

```
#include <rpc/rpc.h>      /* xdr is part of rpc */

xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc,
             writeproc)
   XDR *xdrs;
   u_int sendsize, recvsize;
   char *iohandle;
   int (*readproc)(), (*writeproc)();
```

The routine `xdrrec_create()` provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR

form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX files.

The parameter *xdrs* is similar to the corresponding parameter described previously. The stream does its own data buffering similar to that of standard I/O. The parameters *sendsize* and *recvsize* determine the size in bytes of the output and input buffers respectively. If their values are zero (0), then predetermined defaults are used.

When a buffer needs to be filled or flushed, the routine `readproc()` or `writeproc()` is called respectively. The usage and behavior of these routines are similar to the UNIX system calls `read()` and `write()`. However, the first parameter to each of these routines is the opaque parameter *iohandle*. The other two parameters, and *nbytes* and the results, byte count, are identical to the system routines. If `xxx()` is `readproc()` or `writeproc()`, then it has the following form:

```
/* returns the actual number of bytes transferred. -1 is an error */int
xxx(iohandle, buf, len)
   char *iohandle;
   char *buf;
   int nbytes;
```

The XDR stream provides a means for delimiting records in the byte stream. Abstract data types needed to implement the XDR stream mechanism are discussed in "XDR Stream Implementation" on page 198. The protocol RPC uses to delimit XDR stream records is discussed in "Record-Marking Standard" on page 211.

The primitives that are specific to record streams are:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
   XDR *xdrs;
   bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
   XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
   XDR *xdrs;
```

The routine `xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter *flushnow* is TRUE, then the stream's `writeproc()` is called. Otherwise, `writeproc()` is called when the output buffer is full.

The routine `xdrrec_skiprecord()` causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If no more data is in the stream's input buffer, then the routine `xdrrec_eof()` returns TRUE. That does not mean that no more data is in the underlying file descriptor.

XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

XDR Object

The structure in the following code example defines the interface to an XDR stream.

EXAMPLE 88   XDR Stream Interface Example

```
enum xdr_op {XDR_ENCODE=0, XDR_DECODE=1, XDR_FREE=2};
 typedef struct {
 enum xdr_op x_op;
 struct xdr_ops {
   bool_t (*x_getlong)();        /* get long from stream */
   bool_t (*x_putlong)();        /* put long to stream */
   bool_t (*x_getbytes)();       /* get bytes from stream */
   bool_t (*x_putbytes)();       /* put bytes to stream */
   u_int (*x_getpostn)();        /* return stream offset */
   bool_t (*x_setpostn)();       /* reposition offset */
   caddr_t (*x_inline)();        /* ptr to buffered data */
   VOID (*x_destroy)();          /* free private area */
  bool_t (*x_control)();   /* change, retrieve client info */
  bool_t (*x_getint32)();        /* get int from stream */
   bool_t (*x_putint32)();        /* put intto stream */
  } *x_ops;
  caddr_t x_public;                 /* users' data */
  caddr_t x_private;                /* pointer to private data */
  caddr_t x_base;                   /* private for position info */
  int   x_handy;                 /* extra private word */
 } XDR;
```

The x_op field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value. The fields x_private, x_base, and x_handy are private to the particular stream's implementation. The field x_public is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives. x_getpostn(), x_setpostn(), and x_destroy() are macros for accessing operations.

The operation x_inline() has two parameters: an XDR *, and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. The stream's interpretation is that the bytes in the buffer segment have been consumed. The routine can return NULL if it cannot return a buffer segment of the requested size.

Caution - The `x_inline()` routine is used to squeeze cycles, and the resulting buffer is not data portable. Do not use this feature.

The operations `x_getbytes()` and `x_putbytes()` routinely get and put sequences of bytes from or to the underlying stream. They return `TRUE` if they are successful, and `FALSE` otherwise. The routines have identical parameters (replace *xxx* with the same string in each case.)

```
bool_t
xxxbytes(xdrs, buf, bytecount)
   XDR *xdrs;
   char *buf;
   u_int bytecount;
```

The operations `x_getint32()` and `x_putint32()` receive and put `int` numbers from and to the data stream. These routines are responsible for translating the numbers between the system representation and the (standard) external representation. The UNIX primitives `htonl()` and `ntohl()` can be helpful in accomplishing this objective. The higher-level XDR implementation assumes that signed and unsigned integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return TRUE if they succeed, and FALSE otherwise.

The `x_getint()` and `x_putint()` functions make use of these operations. They have identical parameters:

```
bool_t
xxxint(xdrs, ip)
   XDR *xdrs;
   int32_t *ip;
```

The `long` version of these operations (`x_getlong()` and `x_putlong()`) also call `x_getint32()` and `x_putint32()`, ensuring that a 4–byte quantity is operated on, no matter what system the program is running on.

Implementors of new XDR streams must make an XDR structure with new operation routines available to clients, using some kind of create routine.

Advanced XDR Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists of arbitrary lengths. Unlike the simpler examples covered in the previous sections, the following examples are written using both the XDR C library routines and the XDR data description language. Appendix C, "XDR Protocol Specification" describes this language in detail.

Linked Lists

The "Pointer Example" on page 194 presented a C data structure and its associated XDR routines for an individual's gross assets and liabilities. The following code example uses a linked list to duplicate the pointer example.

EXAMPLE 89   Linked List

```
struct gnumbers {
  int g_assets;
  int g_liabilities;
 };

bool_t
xdr_gnumbers(xdrs, gp)
  XDR *xdrs;
  struct gnumbers *gp;
{
  return(xdr_int(xdrs, &(gp->g_assets) &&
        xdr_int(xdrs, &(gp->g_liabilities))));
}
```

Now assume that you want to implement a linked list of such information. A data structure could be constructed as follows.

```
struct gnumbers_node {
    struct gnumbers gn_numbers;
    struct gnumbers_node *gn_next;
};
typedef struct gnumbers_node *gnumbers_list;
```

Think of the head of the linked list as the data object. That is, the head is not merely a convenient shorthand for a structure. Similarly, the gn_next field is used to indicate whether the object has terminated. Unfortunately, if the object continues, the gn_next field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive declaration of gnumbers_list.

```
struct gnumbers {
    int g_assets;
    int g_liabilities;
};
struct gnumbers_node {
    gnumbers gn_numbers;
    gnumbers_node *gn_next;
};
```

In this description, the Boolean indicates more data follows. If the Boolean is FALSE, it is the last data field of the structure. If it is TRUE, it is followed by a gnumbers structure and, recursively, by a gnumbers_list. Note that the C declaration has no Boolean explicitly declared in it, though the gn_next field implicitly carries the information. The XDR data description has no pointer explicitly declared in it.

Hints for writing the XDR routines for a gnumbers_list follow easily from the preceding XDR description. Note how the primitive xdr_pointer() is used to implement the preceding XDR union.

EXAMPLE 90   xdr_pointer

```
bool_t
xdr_gnumbers_node(xdrs, gn)
  XDR *xdrs;
  gnumbers_node *gn;
{
 return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
        xdr_gnumbers_list(xdrs, &gn->gn_next));
}

bool_t
xdr_gnumbers_list(xdrs, gnp)
  XDR *xdrs;
  gnumbers_list *gnp;
{
 return(xdr_pointer(xdrs, gnp, sizeof(struct gnumbers_node),
                     xdr_gnumbers_node));
 xdr_pointer}
```

The side effect of using XDR on a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This growth is due to the recursion. The following example collapses the last two mutually recursive routines into a single, nonrecursive one.

EXAMPLE 91   Nonrecursive Stack in XDR

```
bool_t
xdr_gnumbers_list(xdrs, gnp)
  XDR *xdrs;
  gnumbers_list *gnp;
{
  bool_t more_data;
  gnumbers_list *nextp;

 for(;;) {
   more_data = (*gnp != NULL);
   if (!xdr_bool(xdrs, &more_data))
    return(FALSE);
```

```
    if (! more_data)
     break;
    if (xdrs->x_op == XDR_FREE)
     nextp = &(*gnp)->gn_next;
    if (!xdr_reference(xdrs, gnp,
    sizeof(struct gnumbers_node), xdr_gnumbers))
    return(FALSE);
    gnp = (xdrs->x_op == XDR_FREE) ? nextp : &(*gnp)->gn_next;
   }
  *gnp = NULL;
  return(TRUE);
}
```

The first task is to find out whether more data exists, so that this Boolean information can be serialized. Notice that this statement is unnecessary in the XDR_DECODE case, because the value of more_data is not known until you deserialize it in the next statement.

The next statement implements XDR on the *more_data* field of the XDR union. Then if no more data exists, you set this last pointer to NULL to indicate the end of the list, and return TRUE because you are done. Note that setting the pointer to NULL is only important in the XDR_DECODE case, because the pointer is already NULL in the XDR_ENCODE and XDR_FREE cases.

Next, if the direction is XDR_FREE, set the value of *nextp* to indicate the location of the next pointer in the list. You set this value now because you need to dereference *gnp* to find the location of the next item in the list. After the next statement, the storage pointed to by *gnp* is freed up and no longer valid. You cannot set this value for all directions, though, because in the XDR_DECODE direction the value of *gnp* is not set until the next statement.

Next, you use XDR on the data in the node using the primitive xdr_reference(). xdr_reference() is like xdr_pointer(), which you used before, but it does not send over the Boolean indicating whether more data exists. You use xdr_reference() instead of xdr_pointer() because you have already used XDR on this information yourself.

Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is xdr_gnumbers(), but each element in the list is actually of type gnumbers_node. You don't pass xdr_gnumbers_node() because it is recursive. Instead, use xdr_gnumbers(), which uses XDR on all of the nonrecursive part. Note that this trick works only if the *gn_numbers* field is the first item in each element, so that their addresses are identical when passed to xdr_reference().

Finally, you update *gnp* to point to the next item in the list. If the direction is XDR_FREE, you set it to the previously saved value. Otherwise, you can dereference *gnp* to get the proper value. Though harder to understand than the recursive version, this nonrecursive routine runs more efficiently because much of the procedure call overhead has been removed. Most lists are small, in the hundreds of items or less, and the recursive version should be sufficient for them.

B

RPC Protocol and Language Specification

This appendix specifies a message protocol used in implementing the RPC package. The message protocol is specified with the XDR language. The companion appendix to this one is Appendix C, "XDR Protocol Specification".

This appendix covers the following topics:

Protocol Overview

The RPC protocol provides for the following:

- Unique specification of a procedure to be called.
- Provisions for matching response messages to request messages.
- Provisions for authenticating the caller to service and the reverse. In addition, the RPC package provides features that detect the following:
  - RPC protocol mismatches
  - Remote program protocol version mismatches
  - Protocol errors, such as incorrect specification of a procedure's parameters
  - Reasons why remote authentication failed

Consider a network file service composed of two programs. One program might handle high-level applications such as file-system access control and locking. The other might handle low-level file I/O and have procedures like read and write. A client of the network file service would call the procedures associated with the two programs of the service on behalf of some user on the client. In the client-server model, a remote procedure call is used to call the service.

RPC Model

The RPC model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location. The caller then transfers control to the procedure, and eventually regains control. At that point, the results of the procedure are extracted from a well-specified location, and the caller continues execution.

The RPC model is similar, in that one thread of control logically winds through two processes. One is the caller's process, the other is a server's process. Conceptually, the caller process sends a call message to the server process and waits for a reply message. The call message contains the procedure's parameters, among other information. The reply message contains the procedure's results, among other information. After the reply message is received, the results of the procedure are extracted, and the caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message.

Note that in this description only one of the two processes is active at any given time. However, the RPC protocol makes no restrictions on the concurrency model implemented. For example, an implementation might choose to have RPC calls be asynchronous, so that the client can do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request so that the server is free to receive other requests.

Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC disregards how a message is passed from one process to another. The protocol handles only specification and interpretation of messages.

RPC does not attempt to ensure transport reliability. Therefore, you must supply the application with information about the type of transport protocol used under RPC. If you tell the RPC service that it is running on top of a reliable transport such as TCP, most of the work is already done for the service. On the other hand, if RPC is running on top of an unreliable transport such as UDP, the service must devise its own retransmission and time-out policy. RPC does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from, but should be explicitly specified by, the underlying transport protocol. For example, suppose RPC is running on top of an unreliable transport. If an application retransmits RPC messages after short timeouts receiving no reply, it can infer only that the procedure was executed zero or more times. If the application does receive a reply, it can infer that the procedure was executed at least once.

A server might choose to remember previously granted requests from a client and not regrant them to ensure some degree of execute-at-most-once semantics. A server can do this by using

the transaction ID that is packaged with every RPC request. The main use of this transaction ID is by the RPC client for matching replies to requests. However, a client application can choose to reuse its previous transaction ID when retransmitting a request. The server application, checking this fact, can choose to remember this ID after granting a request and not regrant requests with the same ID. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a reliable transport such as TCP, the application can infer from a reply message that the procedure was executed exactly once. If the application receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs timeouts and reconnection to handle server crashes.

Binding and Rendezvous Independence

The act of binding a client to a service is not part of the remote procedure call specification. This important and necessary function is left up to some higher-level software. The software can use RPC itself. See "rpcbind Protocol" on page 230.

Implementers should think of the RPC protocol as the jump-subroutine (JSR) instruction of a network. The loader makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the network makes RPC useful, enabling RPC to accomplish this task.

The RPC protocol provides the fields necessary for a client to identify itself to a service and the reverse. Security and access control mechanisms can be built on top of the message authentication. Several different authentication protocols can be supported. A field in the RPC header specifies the protocol being used. You can find more information on authentication protocols in the section "Record-Marking Standard" on page 211.

Program and Procedure Numbers

The RPC call message has three unsigned fields that uniquely identify the procedure to be called:

- Remote program number
- Remote program version number
- Remote procedure number

Program numbers are administered by a central authority, as described in "Program Number Registration" on page 207.

The first implementation of a program most likely has version number 1. Most new protocols evolve into better, stable, and mature protocols. Therefore, a version field of the call message

identifies the version of the protocol that the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the individual program's protocol specification. For example, a file service's protocol specification might state that its procedure number 5 is read and procedure number 12 is write.

Just as remote program protocols can change over several versions, the RPC message protocol itself can change. Therefore, the call message also has in it the RPC version number, which is always equal to 2 for the version of RPC described here.

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

- The remote implementation of RPC does not speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available 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 result is usually a caller-side protocol or programming error.
- The server interprets the parameters to the remote procedure as garbage. Again, this result is usually caused by a disagreement about the protocol between client and service.

Provisions for authentication of caller to service and the reverse are provided as a part of the RPC protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type.

```
enum auth_flavor {
    AUTH_NONE = 0,
    AUTH_SYS = 1,
    AUTH_SHORT = 2,
    AUTH_DES = 3,
  AUTH_KERB = 4
  /* and more to be defined */
  };
  struct opaque_auth {
   enum        auth_flavor;        /* style of credentials */
   caddr_t     oa_base;           /* address of more auth stuff */
   u_int       oa_length;         /* not to exceed MAX_AUTH_BYTES */
  };
```

An `opaque_auth` structure is an `auth_flavor` enumeration followed by bytes that 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. See "Record-Marking Standard" on page 211 for definitions of the various authentication protocols.

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

Program Number Assignment

Program numbers are distributed in groups of `0x20000000`, as shown in the following table.

TABLE 18   RPC Program Number Assignment

Program Numbers	Description
00000000 - 1fffffff	Defined by host
20000000 - 3fffffff	Defined by user
40000000 - 5fffffff	Transient (reserved for customer-written applications)
60000000 - 7fffffff	Reserved
80000000 - 9fffffff	Reserved
a0000000 - bfffffff	Reserved
c0000000 - dfffffff	Reserved
e0000000 - ffffffff	Reserved

Oracle administers the first group of numbers, which should be identical for all customers. If a customer develops an application that might be of general interest, that application should be given an assigned number in the first range.

The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs.

The third group is reserved for applications that generate program numbers dynamically.

The final groups are reserved for future use, and should not be used.

Program Number Registration

RPC program numbers are assigned by the Internet Assigned Numbers Authority (IANA). The policies and procedures for getting a program number assignment are described in Section 13 of RFC 5531. See http://tools.ietf.org/html/rfc5531 for further details. A listing of assigned RPC program numbers can be found at the IANA web site at http://www.iana.org.

Include a compilable `rpcgen .x` file describing your protocol. You are given a unique program number in return.

You can find the RPC program numbers and protocol specifications of standard RPC services in the include files in `/usr/include/rpcsvc`. These services, however, constitute only a small subset of those that have been registered.

Other Uses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which other non-RPC protocols can be implemented. Some of the non-RPC protocols supported by the RPC package are batching and broadcasting.

Batching

Batching enables a client to send an arbitrarily large sequence of call messages to a server. Batching typically uses reliable byte-stream protocols like TCP for its transport. In batching, the client never waits for a reply from the server, and the server does not send replies to batch requests. A sequence of batch calls is usually finished by a non-batch RPC call to flush the pipeline with positive acknowledgement. For more information, see "Batching" on page 97.

Broadcast RPC

In broadcast RPC, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses connectionless, packet-based protocols like UDP as its transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors. Broadcast RPC uses the rpcbind service to achieve its semantics. See "Broadcast RPC" on page 95 and "rpcbind Protocol" on page 230 for further information.

RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top-down style, as shown in the following code example.

EXAMPLE 92   RPC Message Protocol

```
enum msg_type {
  CALL = 0,
  REPLY = 1
};

/*
 * A reply to a call message can take on two forms: The message was
 * either accepted or rejected.
 */
 enum reply_stat {
  MSG_ACCEPTED = 0,
  MSG_DENIED = 1
```

```
 };

/*
 * Given that a call message was accepted, the following is the
 * status of an attempt to call a remote procedure.
 */
enum accept_stat {
  SUCCESS = 0,       /* RPC executed successfully */
  PROG_UNAVAIL = 1,  /* remote service hasn't exported prog */
  PROG_MISMATCH = 2, /* remote service can't support versn # */
  PROC_UNAVAIL = 3,  /* program can't support proc */
  GARBAGE_ARGS = 4   /* procedure can't decode params */
};

/*
 * Reasons a call message was rejected:
 */
enum reject_stat {
  RPC_MISMATCH = 0,  /* RPC version number != 2 */
  AUTH_ERROR = 1     /* remote can't authenticate caller */
};
/*
 * Why authentication failed:
 */
enum auth_stat {
  AUTH_BADCRED = 1,       /* bad credentials */
  AUTH_REJECTEDCRED = 2,  /* clnt must do new session */
  AUTH_BADVERF = 3,       /* bad verifier */
  AUTH_REJECTEDVERF = 4,  /* verif expired or replayed */
  AUTH_TOOWEAK = 5        /* rejected for security */
};

/*
 * The RPC message:
 * All messages start with a transaction identifier, xid, followed
 * by a two-armed discriminated union. The union's discriminant is
 * a msg_type which switches to one of the two types of the
 * message.
 * The xid of a REPLY message always matches that of the
 * initiating CALL message. NB: The xid field is only used for
 * clients matching reply messages with call messages or for servers
 * detecting retransmissions; the service side cannot treat this id as
 * any type of sequence number.
 */
struct rpc_msg {
  unsigned int xid;
  union switch (msg_type mtype) {
   case CALL:
    call_body cbody;
   case REPLY:
    reply_body rbody;
```

```
    } body;
};

/*
 * Body of an RPC request call:
 * In version 2 of the RPC protocol specification, rpcvers must be
 * equal to 2. The fields prog, vers, and proc specify the remote
 * program, its version number, and the procedure within the
 * remote program to be called. After these fields are 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 int rpcvers; /* must be equal to two (2) */
  unsigned int prog;
  unsigned int vers;
  unsigned int proc;
  opaque_auth cred;
  opaque_auth verf;
  /* procedure specific parameters start here */
 };

/*
 * Body of a reply to an RPC request:
 * The call message was either accepted or rejected.
 */
union reply_body switch (reply_stat stat) {
  case MSG_ACCEPTED:
    accepted_reply areply;
  case MSG_DENIED:
    rejected_reply rreply;
} reply;

/*
 * Reply to an RPC request that was accepted by the server: there
 * could be an error even though the request was accepted. The
 * first field is an authentication verifier that the server
 * generates in order to validate itself to the caller. It is
 * followed by a union whose discriminant is an enum accept_stat.
 * The SUCCESS arm of the union is protocol specific.
 * The PROG_UNAVAIL, PROC_UNAVAIL, and GARBAGE_ARGP arms of
 * the union are void. The PROG_MISMATCH arm specifies the lowest
 * and highest version numbers of the remote program supported by
 * the server.
 */
struct accepted_reply {
  opaque_auth verf;
  union switch (accept_stat stat) {
    case SUCCESS:
```

```
   opaque results[0];
    /* procedure-specific results start here */
  case PROG_MISMATCH:
   struct {
    unsigned int low;
    unsigned int high;
   } mismatch_info;
  default:
   /*
    * Void. Cases include PROG_UNAVAIL, PROC_UNAVAIL, and
     * GARBAGE_ARGS.
    */
   void;
 } reply_data;
};

/*
 * Reply to an RPC request that was rejected by the server:
 * The request can be rejected for two reasons: either 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 case of an RPC version mismatch,
 * the server returns the lowest and highest supported RPC
 * version numbers. In case of refused authentication, failure
 * status is returned.
 */
union rejected_reply switch (reject_stat stat) {
  case RPC_MISMATCH:
   struct {
    unsigned int low;
    unsigned int high;
   } mismatch_info;
  case AUTH_ERROR:
   auth_stat stat;
};
```

Record-Marking Standard

When RPC messages are passed on top of a byte-stream transport like TCP, you should try to 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 is 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 the network byte order.

The header encodes two values:

- A Boolean that specifies whether the fragment is the last fragment of the record. Bit value 1 implies the fragment is the last fragment.
- A 31-bit unsigned binary value that is the length in bytes of the fragment's data. The Boolean value is the highest-order bit of the header. The length is the 31 low-order bits. This record specification is not in XDR standard form.

Authentication Protocols

Authentication parameters are opaque but open-ended to the rest of the RPC protocol. This section defines some flavors of authentication that have already been implemented. Other sites are free to invent new authentication types, with the same rules of flavor number assignment for program number assignment. Oracle maintains and administers a range of authentication flavors. To have authentication numbers like RPC program numbers allocated or registered to them, contact the Oracle RPC Administrator.

AUTH_NONE

Calls are often made in which the caller does not authenticate itself and the server disregards who the caller is. In these cases, the *flavor* value of the RPC message's credentials, verifier, and response verifier is `AUTH_NONE`. The flavor value is the "discriminant" of the `opaque_auth` "union." The body length is zero when `AUTH_NONE` authentication flavor is used.

AUTH_SYS

`AUTH_SYS` This is the same as the authentication flavor previously known as `AUTH_UNIX`. The caller of a remote procedure might wish to identify itself using traditional UNIX process permissions authentication. The *flavor* of the `opaque_auth` of such an RPC call message is `AUTH_SYS`. The bytes of the `body` encode the following structure:

```
struct auth_sysparms {
   unsigned int stamp;
   string systemname<255>;
   uid_t uid;
   gid_t gid;
   gid_t gids<10>;
};
```

stamp   is an arbitrary ID that the caller system can generate.

systemname   is the name of the caller's system.

uid   is the caller's effective user ID.

gid   is the caller's effective group ID.

gids   is a counted array of groups in which the caller is a member.

The *flavor* of the verifier accompanying the credentials should be AUTH_NONE.

AUTH_SHORT **Verifier**

When using AUTH_SYS authentication, the *flavor* of the response verifier received in the reply message from the server might be AUTH_NONE or AUTH_SHORT.

If AUTH_SHORT, the bytes of the response verifier's string encode a short_hand_verf structure. This opaque structure can now be passed to the server instead of the original AUTH_SYS credentials.

The server keeps a cache that maps the shorthand opaque structures to the original credentials of the caller. These structures are passed back by way of an AUTH_SHORT style response verifier. The caller can save network bandwidth and server CPU cycles by using the new credentials.

The server can flush the shorthand opaque structure at any time. If a flush occurs, the remote procedure call message is rejected because of an authentication error. The reason for the failure is AUTH_REJECTEDCRED. At this point, the caller might try the original AUTH_SYS style of credentials, as shown in the following figure.

FIGURE 10   Authentication Process Map
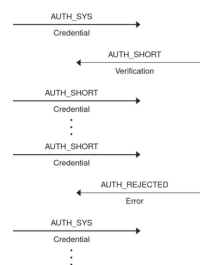
AUTH_DES **Authentication**

You might encounter the following situations with AUTH_SYS authentication:

- Caller identification cannot be guaranteed to be unique if systems with differing operating systems are on the same network.
- No verifier exists, so credentials can easily be faked.

AUTH_DES authentication attempts to fix these two problems.

The first issue is handled by addressing the caller by a simple string of characters instead of by an operating system-specific integer. This string of characters is known as the net name or network name of the caller. The server should not interpret the caller's name in any way other than as the identity of the caller. Thus, net names should be unique for every caller in the naming domain.

Each operating system's implementation of AUTH_DES authentication generates net names for its users that ensure this uniqueness when they call remote servers. Operating systems already distinguish users local to their systems. Extending this mechanism to the network is usually a simple matter.

For example, a user with a user ID of 515 might be assigned the following net name: UNIX. 515@ example .com. This net name contains three items that serve to ensure it is unique. Backtracking, only one naming domain is called example.com in the Internet. Within this domain, only one UNIX user has the user ID 515. However, there might be another user on another operating system, for example VMS, within the same naming domain who, by coincidence, happens to have the same user ID. To ensure that these two users can be distinguished, you add the operating system name. So one user is UNIX.515@ example.com and the other is VMS.515@ example.com.

The first field is actually a naming method rather than an operating system name. It just happens that almost a one-to-one correspondence exists between naming methods and operating systems. If there was a common worldwide naming standard, the first field could be a name from that standard, instead of an operating system name.

AUTH_DES Authentication Verifiers

Unlike AUTH_SYS authentication, AUTH_DES authentication does have a verifier so the server can validate the client's credential, and the reverse. The contents of this verifier are primarily an encrypted timestamp. The server can decrypt this timestamp, and if it is close to its current real time, then the client must have encrypted it correctly. The only way the client could encrypt the timestamp correctly is to know the conversation key of the RPC session. If the client knows the conversation key, it must be the real client.

The conversation key is a DES [5] key that the client generates and notifies the server of in its first RPC call. The conversation key is encrypted using a public-key scheme in this first transaction. The particular public-key scheme used in AUTH_DES authentication is Diffie-Hellman [3] with 192-bit keys. The details of this encryption method are described in "Diffie-Hellman Encryption" on page 217.

The client and the server need the same notion of the current time for the verification to work. If network time synchronization cannot be guaranteed, then the client can synchronize with the server before beginning the conversation. rpcbind provides a procedure, RPCBPROC_GETTIME, which can be used to obtain the current time.

A server can determine if a client timestamp is valid. For any transaction after the first, the server checks for two things:

- The timestamp is greater than the one previously seen from the same client.
- The timestamp has not expired. A timestamp is expired if the server's time is later than the sum of the client's timestamp plus what is known as the client's window. The window is an encrypted number the client passes to the server in its first transaction. The window can be thought of as a lifetime for the credential.

For the first transaction, the server checks that the timestamp has not expired. As an added check, the client sends an encrypted item in the first transaction known as the window verifier. This verifier must be equal to the window minus 1, or the server rejects the credential.

The client must check the verifier returned from the server to be sure it is legitimate. The server sends back to the client the encrypted timestamp it received from the client, minus one second. If the client gets any result other than this one, the verifier is rejected.

Nicknames and Clock Synchronization

After the first transaction, the server's AUTH_DES authentication subsystem returns in its verifier to the client an integer nickname. The client can use this nickname in its further transactions instead of passing its net name, encrypted DES key, and window every time. The nickname is most likely an index into a table on the server that stores for each client its net name, decrypted DES key, and window. It should however be treated as opaque data by the client.

Though originally synchronized, client and server clocks can get out of sync. If this situation occurs, the client RPC subsystem most likely receives an RPC_AUTHERROR at which point it should resynchronize.

A client can still get the RPC_AUTHERROR error even though it is synchronized with the server. The server's nickname table is a limited size, and it can flush entries as needed. The client should resend its original credential and the server gives the client a new nickname. If a server crashes, the entire nickname table is flushed, and all clients have to resend their original credentials.

DES Authentication Protocol (in XDR language)

Credentials are explained in the following example.

EXAMPLE 93   AUTH_DES Authentication Protocol

```
/*
 * There are two kinds of credentials: one in which the client
 * uses its full network name, and one in which it uses its
 * "nickname" (just an unsigned integer) given to it by the
```

```
             * server. The client must use its full name in its first
             * transaction with the server, in which the server returns
             * to the client its nickname. The client may use its nickname
             * in all further transactions with the server. There is no
             * requirement to use the nickname, but it is wise to use it for
             * performance reasons.
             */
            enum authdes_namekind {
              ADN_FULLNAME = 0,
              ADN_NICKNAME = 1
              };

            /*
             * A 64-bit block of encrypted DES data
             */
             typedef opaque des_block[8];

            /*
             * Maximum length of a network user's name
             */
            const MAXNETNAMELEN = 255;

            /*
             * A fullname contains the network name of the client, an
             * encrypted conversation key and the window. The window
             * is actually a lifetime for the credential. If the time
             * indicated in the verifier timestamp plus the window has
             * passed, then the server should expire the request and
             * not grant it. To insure that requests are not replayed,
             * the server should insist that timestamps be greater
             * than the previous one seen, unless it is the first transaction.
             * In the first transaction, the server checks instead that the
             * window verifier is one less than the window.
             */
            struct authdes_fullname {
              string name<MAXNETNAMELEN>; /* name of client */
              des_block key;              /* PK encrypted conversation key */
              unsigned int window;        /* encrypted window */
            };                             /* NOTE: PK means "public key" */

            /*
             * A credential is either a fullname or a nickname
             */
            unionauthdes_credswitch(authdes_namekindadc_namekind){
              case ADN_FULLNAME:
               authdes_fullname adc_fullname;
              case ADN_NICKNAME:
               unsigned int adc_nickname;
            };

            /*
```

```
 * A timestamp encodes the time since midnight, January 1, 1970.
 */
struct timestamp {
  unsigned int seconds;      /* seconds */
  unsigned int useconds;     /* and microseconds */
};

/*
 * Verifier: client variety
  */
struct authdes_verf_clnt {
  timestamp adv_timestamp;   /* encrypted timestamp */
  unsigned int adv_winverf;  /* encrypted window verifier */
};

/*
 * Verifier: server variety
 * The server returns (encrypted) the same timestamp the client gave
 * it minus one second. It also tells the client its nickname to be
 * used in future transactions (unencrypted).
 */
struct authdes_verf_svr {
  timestamp adv_timeverf;    /* encrypted verifier */
  unsigned int adv_nickname; /* new nickname for clnt */};
```

Diffie-Hellman Encryption

In this scheme are two constants, PROOT and HEXMODULUS. The particular values chosen for these constants for the DES authentication protocol are:

```
const PROOT = 3;
const HEXMODULUS = /* hex */
 "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b";
```

The way this scheme works is best explained by an example. Suppose there are two people, A and B, who want to send encrypted messages to each other. A and B each generate a random secret key that they do not disclose to anyone. Let these keys be represented as SK(A) and SK(B). They also publish in a public directory their public keys. These keys are computed as follows:

```
PK(A) = (PROOT ** SK(A)) mod HEXMODULUS
PK(B) = (PROOT ** SK(B)) mod HEXMODULUS
```

The ** notation is used here to represent exponentiation.

Now, both A and B can arrive at the common key between them, represented here as CK(A,B), without disclosing their secret keys.

A computes:

```
CK(A, B) = (PK(B) ** SK(A)) mod HEXMODULUS
```

while B computes:

```
CK(A, B) = (PK(A) ** SK(B)) mod HEXMODULUS
```

These two computations can be shown to be equivalent: `(PK(B)**SK(A)) mod HEXMODULUS` `= (PK(A)**SK(B)) mod HEXMODULUS`. Drop the `mod HEXMODULUS` parts and assume modulo arithmetic to simplify the process:

```
PK(B) ** SK(A) = PK(A) ** SK(B)
```

Then replace `PK(B)` by what B computed earlier and likewise for `PK(A)`.

```
((PROOT ** SK(B)) ** SK(A) = (PROOT ** SK(A)) ** SK(B)
```

which leads to:

```
PROOT ** (SK(A) * SK(B)) = PROOT ** (SK(A) * SK(B))
```

This common key `CK(A,B)` is not used to encrypt the timestamps used in the protocol. It is used only to encrypt a conversation key that is then used to encrypt the timestamps. This approach uses the common key as little as possible, to prevent a break. Breaking the conversation key is a far less serious compromise, because conversations are comparatively short lived.

The conversation key is encrypted using 56-bit DES keys, yet the common key is 192 bits. To reduce the number of bits, 56 bits are selected from the common key as follows. The middle-most 8 bytes are selected from the common key, and then parity is added to the lower-order bit of each byte, producing a 56-bit key with 8 bits of parity.

`AUTH_KERB` Authentication

To avoid compiling Kerberos code into the operating system kernel, the kernel used in the S implementation of `AUTH_KERB` uses a proxy RPC daemon called `kerbd`. The daemon exports three procedures.

- `KGETKCRED` is used by the server-side RPC to check the authenticator presented by the client.
- `KSETKCRED` returns the encrypted ticket and DES session key, given a primary name, instance, and realm.
- `KGETUCRED` is UNIX specific. It returns the user's ID, the group ID, and groups list, assuming that the primary name is mapped to a user name known to the server.

The best way to describe how Kerberos works is to use an example based on a service currently implementing Kerberos: the network file system (NFS). The NFS service on server *s* is assumed to have the well-known principal name nfs.*s*. A privileged user on client *c* is assumed to have the primary name `root` and an instance *c*. Note that, unlike `AUTH_DES`, when the user's ticket-granting ticket has expired, `kinit()` must be reinvoked. NFS service for Kerberos mounts fail until a new ticket-granting ticket is obtained.

NFS Mount Example

This section follows an NFS mount request from start to finish using `AUTH_KERB`. Because mount requests are executed as root, the user's identity is root.*c*.

Client *c* makes a `MOUNTPROC_MOUNT` request to the server *s* to obtain the file handle for the directory to be mounted. The client mount program makes an NFS mount system call, handing the client kernel the file handle, mount flavor, time synchronization address, and the server's well-known name, `nfs.`*s*. Next the client kernel contacts the server at the time synchronization host to obtain the client-server time bias.

The client kernel makes the following RPC calls.

1. `KSETKCRED` to the local `kerbd` to obtain the ticket and session key.
2. `NFSPROC_GETATTR` to the server's NFS service, using the full name credential and verifier. The server receives the calls and makes the `KGETKCRED` call to its local `kerbd` to check the client's ticket.

The server's `kerbd` and the Kerberos library decrypt the ticket and return, among other data, the principal name and DES session key. The server checks that the ticket is still valid, uses the session key to decrypt the DES-encrypted portions of the credential and verifier, and checks that the verifier is valid.

The possible Kerberos authentication errors returned at this time are:

- `AUTH_BADCRED` is returned if the verifier is invalid because the decrypted *win* in the credential and *win +1* in the verifier do not match, or the timestamp is not within the window range.
- `AUTH_REJECTEDCRED` is returned if a replay is detected.
- `AUTH_BADVERF` is returned if the verifier is garbled.

If no errors are received, the server caches the client's identity and allocates a nickname, which is a small integer, to be returned in the NFS reply. The server then checks if the client is in the same realm as the server. If so, the server calls `KGETUCRED` to its local `kerbd` to translate the principal's primary name into UNIX credentials. If the previous name is not translatable, the user is marked anonymous. The server checks these credentials against the file system's export information. Consider these three cases:

1. If the `KGETUCRED` call fails and anonymous requests are allowed, the UNIX credentials of the anonymous user are assigned.
2. If the `KGETUCRED` call fails and anonymous requests are not allowed, the NFS call fails with the `AUTH_TOOWEAK`.
3. If the `KGETUCRED` call succeeds, the credentials are assigned, and normal protection checking follows, including checking for root permission.

Next, the server sends an NFS reply, including the nickname and server's verifier. The client receives the reply, decrypts and validates the verifier, and stores the nickname for future calls. The client makes a second NFS call to the server, and the calls to the server described

previously are repeated. The client kernel makes an `NFSPROC_STATVFS` call to the server's NFS service, using the nickname credential and verifier described previously. The server receives the call and validates the nickname. If it is out of range, the error `AUTH_BADCRED` is returned. The server uses the session key just obtained to decrypt the DES-encrypted portions of the verifier and validates the verifier.

The possible Kerberos authentication errors returned at this time are:

- `AUTH_REJECTEDVERF`, which is returned if the timestamp is invalid, a replay is detected, or if the timestamp is not within the window range
- `AUTH_TIMEEXPIRE`, which is returned if the service ticket is expired

If no errors are received, the server uses the nickname to retrieve the caller's UNIX credentials. Then it checks these credentials against the file system's export information, and sends an NFS reply that includes the nickname and the server's verifier. The client receives the reply, decrypts and validates the verifier, and stores the nickname for future calls. Last, the client's NFS mount system call returns, and the request is finished.

KERB Authentication Protocol

The following example of `AUTH_KERB` has many similarities to the one for `AUTH_DES`, shown in the following code example. Note the differences.

EXAMPLE 94   AUTH_KERB Authentication Protocol

```
#define AUTH_KERB 4
/*
 * There are two kinds of credentials: one in which the client
 * sends the (previously encrypted)
Kerberos ticket, and one in
 * which it uses its "nickname" (just an unsigned integer)
 * given to it by the server. The client must use its full name
 * in its first transaction with the server, in which the server
 * returns to the client its nickname. The client may use
 * its nickname in all further transactions with the server
 * (until the ticket expires). There is no requirement to use
 * the nickname, but it is wise to use it for performance reasons.
 */
enum authkerb_namekind {
  AKN_FULLNAME = 0,
  AKN_NICKNAME = 1
 };

/*
 * A fullname contains the encrypted service ticket and the
 * window.  The window is actually a lifetime
 * for the credential. If the time indicated in the verifier
 * timestamp plus the window has passed, then the server should
```

```
 * expire the request and not grant it. To insure that requests
 * are not replayed, the server should insist that timestamps be
 * greater than the previous one seen, unless it is the first
 * transaction. In the first transaction, the server checks
 * instead that the window verifier is one less than the window.
 */
struct authkerb_fullname {
  KTEXT_ST ticket;              /* Kerberos service ticket */
  unsigned long window;        /* encrypted window */
};
/*
 * A credential is either a fullname or a nickname
 */
union authkerb_credswitch(authkerb_namekind akc_namekind){
  case AKN_FULLNAME:
   authkerb_fullname akc_fullname;
  case AKN_NICKNAME:
   unsigned long akc_nickname;
};

/*
 * A timestamp encodes the time since midnight, January 1, 1970.
 */
struct timestamp {
  unsigned long seconds;      /* seconds */
  unsigned long useconds;     /* and microseconds */
};

/*
 * Verifier: client variety

 */
struct authkerb_verf_clnt {
  timestamp akv_timestamp;   /* encrypted timestamp */
  unsigned long akv_winverf;  /* encrypted window verifier */
};

/*
 * Verifier: server variety
 * The server returns (encrypted) the same timestamp the client
 * gave it minus one second. It also tells the client its
 * nickname to be used
in future transactions (unencrypted).
 */
struct authkerb_verf_svr {
  timestamp akv_timeverf;    /* encrypted verifier */
  unsigned long akv_nickname; /* new nickname for clnt */
};
```

RPC Language Specification

Just as the XDR data types needed to be described in a formal language, the procedures that operate on these XDR data types in a formal language needed to be described. The RPC Language, an extension to the XDR language, serves this purpose. The following example is used to describe the essence of the language.

Example Service Described in the RPC Language

The following code example shows the specification of a simple ping program.

EXAMPLE 95   ping Service Using RPC Language

```
/*
 * Simple ping program
 */
program PING_PROG {
  version PING_VERS_PINGBACK {
   void
   PINGPROC_NULL(void) = 0;
   /*
   * ping the caller, return the round-trip time
   * in milliseconds. Return a minus one (-1) if
    * operation times-out
   */
  int
   PINGPROC_PINGBACK(void) = 1;
   /* void - above is an argument to the call */
  } = 2;
/*
 * Original version
 */
  version PING_VERS_ORIG {
   void
   PINGPROC_NULL(void) = 0;
  } = 1;
} = 200000;
const PING_VERS = 2; /* latest version */
```

The first version described is `PING_VERS_PINGBACK` with two procedures, `PINGPROC_NULL` and `PINGPROC_PINGBACK`.

`PINGPROC_NULL` takes no arguments and returns no results, but it is useful for such things as computing round-trip times from the client to the server and back again. By convention, procedure `0` of any RPC program should have the same semantics, and never require authentication.

The second procedure returns the amount of time in microseconds that the operation used.

The next version, `PING_VERS_ORIG`, is the original version of the protocol and does not contain the `PINGPROC_PINGBACK` procedure. It is useful for compatibility with old client programs.

RPCL Syntax

The RPC language (RPCL) is similar to C. This section describes the syntax of the RPC language, and includes examples. It also shows how RPC and XDR type definitions are compiled into C type definitions in the output header file.

An RPC language file consists of a series of definitions.

```
definition-list:
    definition;
    definition; definition-list
```

The file recognizes six types of definitions:

```
definition:
      enum-definition
   const-definition
   typedef-definition
   struct-definition
   union-definition
   program-definition
```

Definitions are not the same as declarations. No space is allocated by a definition, only the type definition of a single or series of data elements. This behavior means that variables still must be declared.

The RPC language is identical to the XDR language, except for the added definitions described in the following table.

TABLE 19   RPC Language Definitions

Term	Definition
program-definition	`program program-ident {version-list} = value`
version-list	`version;`
	`version; version-list`
version	`version version-ident {procedure-list} = value`
procedure-list	`procedure;`
	`procedure; procedure-list`
procedure	`type-ident procedure-ident (type-ident) = value`

In the RPC language:

- The following keywords are added and cannot be used as identifiers:

  `program version.`
- Neither version name nor a version number can occur more than once within the scope of a program definition.
- Neither a procedure name nor a procedure number can occur more than once within the scope of a version definition.
- Program identifiers are in the same namespace as constant and type identifiers.
- Only unsigned constants can be assigned to programs, versions, and procedures.

RPCL Enumerations

RPC/XDR enumerations have a similar syntax to C enumerations.

```
enum-definition:
  "enum" enum-ident "{"
   enum-value-list
  "}"
enum-value-list:
  enum-value
  enum-value "," enum-value-list
enum-value:
  enum-value-ident
  enum-value-ident "=" value
```

Here is an example of an XDR enum and the C enum to which it gets compiled.

```
enum colortype {              enum colortype {
  RED = 0,                       RED = 0,
  GREEN = 1,        -->          GREEN = 1,
  BLUE = 2                       BLUE = 2,
};                            };
                              typedef enum colortype colortype;
```

RPCL Constants

You can use XDR symbolic constants wherever an integer constant is used. A typical use might be in array size specifications:

```
    const-definition:
    const const-ident = integer
```

The following example defines a constant, `DOZEN`, as equal to 12.

```
const DOZEN = 12; --> #define DOZEN 12
```

RPCL Type Definitions

XDR `typedefs` have the same syntax as C `typedefs`.

```
typedef-definition:
      typedef declaration
```

This example defines an `fname_type` used for declaring file-name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

RPCL Declarations

XDR has four kinds of declarations. These declarations must be a part of a `struct` or a `typedef`. They cannot stand alone.

```
declaration:
      simple-declaration
   fixed-array-declaration
   variable-array-declaration
   pointer-declaration
```

RPCL Simple Declarations

Simple declarations are just like simple C declarations.

```
simple-declaration:
   type-ident variable-ident
```

Example:

```
   colortype color; --> colortype color;
```

RPCL Fixed-Length Array Declarations

Fixed-length array declarations are just like C array declarations.

```
fixed-array-declaration:
      type-ident variable-ident [value]
```

Example:

```
colortype palette[8]; --> colortype palette[8];
```

Many programmers confuse variable declarations with type declarations. Note that `rpcgen` does not support variable declarations. The following example is a program that does not compile.

```
int data[10];
program P {
   version V {
      int PROC(data) = 1;
  } = 1;
} = 0x200000;
```

The previous example does not compile because of the variable declaration:

```
int data[10]
```

Instead use:

```
typedef int data[10];
```

or

```
struct data {int dummy [10]};
```

RPCL Variable-Length Array Declarations

Variable-length array declarations have no explicit syntax in C. The XDR language does have a syntax, using angle brackets:

```
variable-array-declaration:
      type-ident variable-ident <value>
      type-ident variable-ident < >
```

The maximum size is specified between the angle brackets. You can omit the size, indicating that the array can be of any size.

```
int heights<12>; /* at most 12 items */
int widths<>; /* any number of items */
```

Because variable-length arrays have no explicit syntax in C, these declarations are compiled into `struct` declarations. An example is the `heights` declaration compiled into the following `struct`.

```
struct {
  u_int heights_len;                  /* # of items in array */
  int *heights_val;               /* pointer to array */
} heights;
```

The number of items in the array is stored in the _len component and the pointer to the array is stored in the _val component. The first part of each component name is the same as the name of the declared XDR variable, *heights*.

RPCL Pointer Declarations

Pointer declarations are made in XDR exactly as they are in C. Address pointers are not really sent over the network. Instead, XDR pointers are useful for sending recursive data types such as lists and trees. The type is called `optional-data`, not `pointer`, in XDR language.

```
pointer-declaration:
  type-ident *variable-ident
```

Example:

```
listitem *next; --> listitem *next;
```

RPCL Structures

An RPC/XDR `struct` is declared almost exactly like its C counterpart. It looks like the following.

```
struct-definition:
  struct struct-ident "{"
      declaration-list
  "}"

declaration-list:
  declaration ";"
  declaration ";" declaration-list
```

The following XDR structure is an example of a 2–D coordinate and the C structure that it compiles into.

```
struct coord {              struct coord {
  int x;          -->           int x;
  int y;                        int y;
};                          };
                            typedef struct coord coord;
```

The output is identical to the input, except for the added `typedef` at the end of the output. This `typedef` enables you to use `coord` instead of `struct coord` when declaring items.

RPCL Unions

XDR unions are discriminated unions, and do not look like C unions. They are more similar to Pascal variant records.

```
union-definition:
 "union" union-ident "switch" "("simple declaration")" "{"
   case-list
  "}"
```

```
case-list:
  "case" value ":" declaration ";"
  "case" value ":" declaration ";" case-list
  "default" ":" declaration ";"
```

The following example is of a type returned as the result of a "read data" operation: if no error occurs, return a block of data. Otherwise, don't return anything.

```
union read_result switch (int errno) {
  case 0:
   opaque data[1024];
  default:
   void;
  };
```

This union compiles into the following:

```
struct read_result {
  int errno;
  union {
   char data[1024];
  } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output `struct` has the same name as the type name, except for the trailing _u.

RPCL Programs

You declare RPC programs using the following syntax:

```
program-definition:
  "program" program-ident "{"
   version-list
  "}" "=" value;
version-list:
  version ";"
  version ";" version-list
version:
  "version" version-ident "{"
   procedure-list
  "}" "=" value;
procedure-list:
  procedure ";"
  procedure ";" procedure-list
procedure:
  type-ident procedure-ident "(" type-ident ")" "=" value;
```

When the -N option is specified, `rpcgen` also recognizes the following syntax.

```
procedure:
  type-ident procedure-ident "(" type-ident-list ")" "=" value;
type-ident-list:
  type-ident
  type-ident "," type-ident-list
```

Example:

```
/*
 * time.x: Get or set the time. Time is represented as seconds
 * since 0:00, January 1, 1970.
 */
program TIMEPROG {
   version TIMEVERS {
      unsigned int TIMEGET(void) = 1;
   void TIMESET(unsigned) = 2;
  } = 1;
} = 0x20000044;
```

Note that the void argument type means that no argument is passed.

The following file compiles into these #define statements in the output header file.

```
#define TIMEPROG 0x20000044
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

RPCL Special Cases

Several exceptions to the RPC language rules follow.

RPCL C-style Mode

The features of the C-style mode of rpcgen have implications for the passing of void arguments. No arguments need be passed if their value is void.

RPCL Booleans

C has no built-in Boolean type. However, the RPC library uses a Boolean type called bool_t that is either TRUE or FALSE. Parameters declared as type bool in XDR language are compiled into bool_t in the output header file.

Example:

```
bool married; --> bool_t married;
```

RPCL Strings

The C language has no built-in string type, but instead uses the null-terminated `char *` convention. In C, strings are usually treated as null-terminated single-dimensional arrays.

In XDR language, strings are declared using the `string` keyword, and compiled into type `char *` in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings, not counting the `NULL` character. You can omit the maximum size, indicating a string of arbitrary length.

Examples:

```
string name<32>;   --> char *name;
string longname<>; --> char *longname;
```

`NULL` strings cannot be passed; however, a zero-length string (that is, just the terminator or `NULL` byte) can be passed.

RPCL Opaque Data

Opaque data is used in XDR to describe untyped data, that is, sequences of arbitrary bytes. You can declare opaque data either as a fixed-length or variable-length array.

Examples:

```
opaque diskblock[512]; --> char diskblock[512];
opaque filedata<1024>; --> struct {
                                u_int filedata_len;
                                char *filedata_val;
                            } filedata;
```

RPCL Voids

In a void declaration, the variable is not named. The declaration is just `void` and nothing else. Void declarations can only occur in two places: union definitions and program definitions as the argument or result of a remote procedure; for example, no arguments are passed.

`rpcbind` Protocol

`rpcbind` maps RPC program and version numbers to universal addresses, thus making dynamic binding of remote programs possible.

`rpcbind` is bound to a well-known address of each supported transport, and other programs register their dynamically allocated transport addresses with it. `rpcbind` then makes those

addresses publicly available. Universal addresses are string representations of the transport-dependent address. They are defined by the addressing authority of the given transport.

rpcbind also aids in broadcast RPC. RPC programs have different addresses on different systems, so direct broadcasts to all these programs are not possible. rpcbind, however, has a well-known address. So, to broadcast to a given program, the client sends its message to the rpcbind process on the system it chooses to reach. rpcbind picks up the broadcast and calls the local service specified by the client. When rpcbind gets a reply from the local service, it passes the reply on to the client.

The following code example shows the rpcbind Protocol Specification in RPC Language.

EXAMPLE 96   rpcbind Protocol Specification in RPC Language

```
/*
 * rpcb_prot.x
 * RPCBIND protocol in rpc language
 */
/*
 * A mapping of (program, version, network ID) to universal
address
 */
struct rpcb {
 rpcproc_t r_prog;           /* program number */
 rpcvers_t r_vers;           /* version number */
 string r_netid<>;             /* network id */
 string r_addr<>;              /* universal address */
 string r_owner<>;             /* owner of this service */ };
/* A list of mappings */
struct rpcblist {
 rpcb rpcb_map;
 struct rpcblist *rpcb_next;
};

/* Arguments of remote calls */
struct rpcb_rmtcallargs {
 rpcprog_t prog;             /* program number */
 rpcvers_t vers;             /* version number */
 rpcproc_t proc;             /* procedure number */
 opaque args<>;                /* argument */
};

/* Results of the remote call */
struct rpcb_rmtcallres {
 string addr<>;                /* remote universal address */
 opaque results<>;             /* result */
};

/*
 * rpcb_entry contains a merged address of a service on a
```

```
            particular
             * transport, plus associated netconfig information. A list of
             * rpcb_entrys is returned by RPCBPROC_GETADDRLIST. See
            netconfig.h
             * for values used in r_nc_* fields.
             */
            struct rpcb_entry {
             string          r_maddr<>;       /* merged address of service */
             string          r_nc_netid<>;   /* netid field */
             unsigned int    r_nc_semantics; /* semantics of transport */
             string          r_nc_protofmly<>; /* protocol family */
             string          r_nc_proto<>;   /* protocol name */
            };

            /* A list of addresses supported by a service. */
            struct rpcb_entry_list {
             rpcb_entry rpcb_entry_map;
             struct rpcb_entry_list *rpcb_entry_next;
            };

            typedef rpcb_entry_list *rpcb_entry_list_ptr;

            /* rpcbind statistics */
            const rpcb_highproc_2 = RPCBPROC_CALLIT;
            const rpcb_highproc_3 = RPCBPROC_TADDR2UADDR;
            const rpcb_highproc_4 = RPCBPROC_GETSTAT;
            const RPCBSTAT_HIGHPROC = 13;  /* # of procs in rpcbind V4 plus
            one */
            const RPCBVERS_STAT = 3;  /* provide only for rpcbind V2, V3 and
            V4 */
            const RPCBVERS_4_STAT = 2;
            const RPCBVERS_3_STAT = 1;
            const RPCBVERS_2_STAT = 0;

            /* Link list of all the stats about getport and getaddr */
            struct rpcbs_addrlist {
             rpcprog_t prog;
             rpcvers_t vers;
             int success;
             int failure;
             string netid<>;
             struct rpcbs_addrlist *next;
            };

            /* Link list of all the stats about rmtcall */
            struct rpcbs_rmtcalllist {
             rpcprog_t prog;
             rpcvers_t vers;
             rpcproc_t proc;
             int success;
             int failure;
```

```
 int indirect;   /* whether callit or indirect */
 string netid<>;
 struct rpcbs_rmtcalllist *next;
};


typedef int rpcbs_proc[RPCBSTAT_HIGHPROC];
typedef rpcbs_addrlist *rpcbs_addrlist_ptr;
typedef rpcbs_rmtcalllist *rpcbs_rmtcalllist_ptr;

struct rpcb_stat {
 rpcbs_proc              info;
 int                     setinfo;
 int                     unsetinfo;
 rpcbs_addrlist_ptr      addrinfo;
 rpcbs_rmtcalllist_ptr   rmtinfo;
};

/*
 * One rpcb_stat structure is returned for each version of rpcbind
 * being monitored.
 */
typedef rpcb_stat rpcb_stat_byvers[RPCBVERS_STAT];
/* rpcbind procedures */
program RPCBPROG {
 version RPCBVERS {
  void
  RPCBPROC_NULL(void) = 0;

  /*
   * Registers the tuple [r_prog, r_vers, r_addr, r_owner,

   * r_netid]. The rpcbind server accepts requests for this
   * procedure on only the loopback transport for security
   * reasons. Returns TRUE if successful, FALSE on failure.
   */
  bool
  RPCBPROC_SET(rpcb) = 1;

  /*
   * Unregisters the tuple [r_prog, r_vers, r_owner, r_netid].

   * If vers is zero, all versions are
unregistered. The rpcbind
   * server accepts requests for this procedure on only the
   * loopback transport for security reasons.  Returns TRUE if
   * successful, FALSE on failure.
   */
  bool
  RPCBPROC_UNSET(rpcb) = 2;

  /*
```

```
             * Returns the universal address where the triple [r_prog,

             * r_vers, r_netid] is registered.  If r_addr specified,
             * return a universal address merged on r_addr. Ignores
             * r_owner. Returns FALSE on failure.
             */
            string
            RPCBPROC_GETADDR(rpcb) = 3;

            /* Returns a list of all mappings. */

          rpcblist
           RPCBPROC_DUMP(void) = 4;

           /*
            * Calls the procedure on the remote system.  If it is not

            * registered, this procedure IS quiet; that is, it DOES NOT
            * return error information.
            */
           rpcb_rmtcallres
           RPCBPROC_CALLIT(rpcb_rmtcallargs) = 5;

           /*
            * Returns the time on the rpcbind server's system.

            */
           unsigned int
           RPCBPROC_GETTIME(void) = 6;

           struct netbuf
           RPCBPROC_UADDR2TADDR(string) = 7;


           string
           RPCBPROC_TADDR2UADDR(struct netbuf) = 8;

           } = 3;
           version RPCBVERS4 {
           bool
           RPCBPROC_SET(rpcb) = 1;

           bool
           RPCBPROC_UNSET(rpcb) = 2;

           string
           RPCBPROC_GETADDR(rpcb) = 3;

           rpcblist_ptr
           RPCBPROC_DUMP(void) = 4;
```

```
   /*
    * NOTE: RPCBPROC_BCAST has the same functionality as CALLIT;

    * the new name is
intended to indicate that this procedure
    * should be used for broadcast RPC, and RPCBPROC_INDIRECT
    * should be used for indirect calls.
    */
   rpcb_rmtcallres
   RPCBPROC_BCAST(rpcb_rmtcallargs) = RPCBPROC_CALLIT;

   unsigned int
   RPCBPROC_GETTIME(void) = 6;

   struct netbuf
   RPCBPROC_UADDR2TADDR(string) = 7;


   string
   RPCBPROC_TADDR2UADDR(struct netbuf) = 8;


   /*
    * Same as RPCBPROC_GETADDR except that if the given version

    * number is not available, the address is not returned.
    */
   string
   RPCBPROC_GETVERSADDR(rpcb) = 9;


   /*
    * Calls the procedure on the remote system.  If it is not
    * registered, this procedure IS NOT quiet; that is, it DOES
    * return error information.
    */
   rpcb_rmtcallres
   RPCBPROC_INDIRECT(rpcb_rmtcallargs) = 10;

   /*
    * Same as RPCBPROC_GETADDR except that it returns a list of

    * addresses registered for the combination (prog, vers).
    */
   rpcb_entry_list_ptr
   RPCBPROC_GETADDRLIST(rpcb) = 11;

   /*
    * Returns statistics about the rpcbind server's activity.

    */
```

```
  rpcb_stat_byvers
  RPCBPROC_GETSTAT(void) = 12;
 } = 4;
} = 100000;
```

`rpcbind` Operation

rpcbind is contacted by way of an assigned address specific to the transport being used. For TCP/IP and UDP/IP, for example, it is port number 111. Each transport has such an assigned well-known address. This section describes a description of each of the procedures supported by rpcbind.

RPCBPROC_NULL   This procedure does no work. By convention, procedure zero of any program takes no parameters and returns no results.

RPCBPROC_SET   When a program first becomes available on a system, it registers itself with the rpcbind program running on the same system. The program passes its program number *prog*, version number *vers*, network identifier *netid*, and the universal address *uaddr*; on which it awaits service requests.

The procedure returns a Boolean response with the value TRUE if the procedure successfully established the mapping and FALSE otherwise. The procedure refuses to establish a mapping if one already exists for the ordered set (*prog*, *vers*, *netid*).

Neither *netid* nor *uaddr* can be NULL, and that *netid* should be a valid network identifier on the system making the call.

RPCBPROC_UNSET   When a program becomes unavailable, it should unregister itself with the rpcbind program on the same system.

The parameters and results have meanings identical to those of RPCBPROC_SET. The mapping of the (*prog*, *vers*, *netid*) tuple with *uaddr* is deleted.

If *netid* is NULL, all mappings specified by the ordered set (*prog*, *vers*, *) and the corresponding universal addresses are deleted. Only the owner of the service or the superuser is allowed to unset a service.

RPCBPROC_GETADDR   Given a program number *prog*, version number *vers*, and network identifier *netid*, this procedure returns the universal address on which the program is awaiting call requests.

The *netid* field of the argument is ignored and the *netid* is inferred from the *netid* of the transport on which the request came in.

RPCBPROC_DUMP   This procedure lists all entries in rpcbind's database.

The procedure takes no parameters and returns a list of program, version, netid, and universal addresses. Call this procedure using a stream rather than a datagram transport to avoid the return of a large amount of data.

RPCBPROC_CALLIT   This procedure enables a caller to call another remote procedure on the same system without knowing the remote procedure's universal address. `RPCBPROC_CALLIT` support broadcasts to arbitrary remote programs through `rpcbind`'s universal address.

The parameters *prog, vers, proc*, and the *args_ptr* are the program number, version number, procedure number, and parameters of the remote procedure.

This procedure sends a response only if the procedure was successfully executed, and is silent (no response) otherwise.

The procedure returns the remote program's universal address, and the results of the remote procedure.

RPCBPROC_GETTIME   This procedure returns the local time on its own system in seconds since midnight of January 1, 1970.

RPCBPROC_UADDR2TADDR   This procedure converts universal addresses to transport (`netbuf`) addresses. `RPCBPROC_UADDR2TADDR` is equivalent to `uaddr2taddr()`. See the `netdir`(3NSL) man page. Only processes that cannot link to the name-to-address library modules should use `RPCBPROC_UADDR2TADDR`.

RPCBPROC_TADDR2UADDR   This procedure converts transport (`netbuf`) addresses to universal addresses. `RPCBPROC_TADDR2UADDR` is equivalent to `taddr2uaddr()`. See the `netdir`(3NSL) man page. Only processes that cannot link to the name-to-address library modules should use `RPCBPROC_TADDR2UADDR`.

Version 4 rpcbind   Version 4 of the `rpcbind` protocol includes all of the previous procedures, and adds several others.

RPCBPROC_BCAST   This procedure is identical to the version 3 `RPCBPROC_CALLIT` procedure. The new name indicates that the procedure should be used for broadcast RPCs only. `RPCBPROC_INDIRECT`, defined in the following text, should be used for indirect RPC calls.

RPCBPROC_GETVERSADDR   This procedure is similar to `RPCBPROC_GETADDR`. The difference is that the `r_vers` field of the `rpcb` structure can be used to specify the version of interest. If that version is not registered, no address is returned.

RPCBPROC_INDIRECT   This procedure is similar to `RPCBPROC_CALLIT`. Instead of being silent about errors, such as the program not being registered on the system, this procedure returns an indication of the error. Do not use this procedure for broadcast RPC. Use it with indirect RPC calls only.

RPCBPROC_GETADDRLIST   This procedure returns a list of addresses for the given *rpcb* entry. The client might be able to use the results to determine alternate transports that it can use to communicate with the server.

RPCBPROC_GETSTAT   This procedure returns statistics on the activity of the rpcbind server. The information lists the number and kind of requests the server has received.

All procedures except RPCBPROC_SET and RPCBPROC_UNSET can be called by clients running on a system other than a system on which rpcbind is running. rpcbind accepts only RPCPROC_SET and RPCPROC_UNSET requests on the loopback transport.

C

XDR Protocol Specification

This appendix contains the XDR Protocol Language Specification. It covers the following topics:

-
-
-

XDR Protocol Introduction

External data representation (XDR) is a standard for the description and encoding of data. The XDR protocol is useful for transferring data between different computer architectures and has been used to communicate data between very diverse systems. XDR fits into the ISO reference model's presentation layer (layer 6) and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between the two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats and can only be used to describe data. It is not a programming language. This language enables you to describe intricate data formats in a concise manner. The XDR language is similar to the C language. Protocols such as RPC and NFS use XDR to describe the format of their data.

The XDR standard assumes that bytes, or octets, are portable and that a byte is defined to be 8 bits of data.

Graphic Box Notation

This appendix uses graphic box notation for illustration and comparison. In most illustrations, each box depicts a byte. The representation of all items requires a multiple of 4 bytes (or 32 bits) of data. The bytes are numbered `0` through `n  -1`. The bytes are read or written to some byte stream such that byte `m` always precedes byte `m+1`. The `n` bytes are followed by enough (0 to 3) residual zero bytes, `r`, to make the total byte count a multiple of four. Ellipses (. . .) between boxes show zero or more additional bytes where required, as shown in the following illustration.
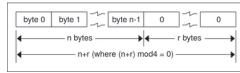
Basic Block Size

Choosing the XDR block size requires a tradeoff. Choosing a small size such as 2 makes the encoded data small, but causes alignment problems for systems that are not aligned on these boundaries. A large size such as 8 means the data is aligned on virtually every system, but causes the encoded data to grow too large. Four was chosen as a compromise. Four is big enough to support most architectures efficiently.

This basic block size of 4 does not mean that the computers cannot utilise standard `XDR`, just that they do so at a greater overhead per data item than 4-byte (32-bit) architectures. Four is also small enough to keep the encoded data restricted to a reasonable size.

The same data should encode into an equivalent result on all systems so that encoded data can be compared or checksummed. So, variable-length data must be padded with trailing zeros.

XDR Data Type Declarations

Each of the sections that follow:

- Describes a data type defined in the XDR standard
- Shows how that data type is declared in the language
- Includes a graphic illustration of the encoding

For each data type in the language a general paradigm declaration is shown. Note that angle brackets (`<` and `>`) denote variable-length sequences of data and square brackets (`[`and`]`) denote fixed-length sequences of data. `n`, `m`, and `r` denote integers. For the full language specification, refer to .

Some data types include specific examples. A more extensive example is given in the section .

Signed Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648,2147483647]. The integer is represented in two's complement notation; the most and least significant bytes are 0 and 3, respectively.

Declaration

Integers are declared:

```
int identifier;
```

Signed Integer Encoding

Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0, 4294967295]. The integer is represented by an unsigned binary number that has most and least-significant bytes of 0 and 3 respectively.

Declaration

An unsigned integer is declared as follows.

```
unsigned int identifier;
```

Unsigned Integer Encoding

Enumerations

Enumerations have the same representation as signed integers and are handy for describing subsets of the integers. The encoding for enumerations is the same as shown in "Signed Integer Encoding" on page 241.

Enumerated data is declared as follows.

```
enum {name-identifier = constant, ... } identifier;
```

For example, an enumerated type could represent the three colors red, yellow, and blue as follows.

```
enum {RED = 2, YELLOW = 3, BLUE = 5} colors;
```

Do not assign to an `enum` an integer that has not been assigned in the `enum` declaration.

Booleans

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are integers of value `0` or `1`. The encoding for Booleans is the same as show in .

Booleans are declared as follows.

```
bool identifier;
```

This is equivalent to:

```
enum {FALSE = 0, TRUE = 1} identifier;
```

Hyper Integer and Unsigned Hyper Integer

The standard defines 64–bit (8-byte) numbers called `hyper int` and `unsigned hyper int` with representations that are the obvious extensions of `integer` and `unsigned integer`, defined previously. They are represented in two's complement notation; the most-significant and least-significant bytes are 0 and 7, respectively.

Declaration

Hyper integers are declared as follows.

```
hyper int  identifier;
unsigned hyper int identifier;
```

Hyper Integer Encoding
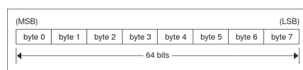
Floating Point

The standard defines the floating-point data type `float` (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers [1]. The following three fields describe the single-precision floating-point number:

`S`: The sign of the number. Values 0 and 1 represent positive and negative respectively. One bit.

`E`: The exponent of the number, base 2. Eight bits are in this field. The exponent is biased by 127.

`F`: The fractional part of the number's mantissa, base 2. Twenty-three bits are in this field.

Therefore, the floating-point number is described by.

```
(-1)**S * 2**(E-Bias) * 1.F
```

Declaration

Single-precision floating-point data is declared as follows.

```
float identifier;
```

Double-precision floating-point data is declared as follows.

```
double identifier;
```

Double-Precision Floating Point Encoding
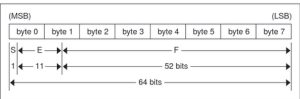
Just as the most and least significant bytes of an integer are 0 and 3, the most-significant and least-significant bits of a double-precision floating-point number are 0 and 63. The beginning bit, and most significant bit, offsets of S, E, and F are 0, 1, and 12 respectively.

These offsets refer to the logical positions of the bits, not to their physical locations, which vary from medium to medium.

Consult the IEEE specifications about the encoding for signed zero, signed infinity (overflow), and de-normalized numbers (underflow) [1]. According to IEEE specifications, the `NaN` (not a number) is system dependent and should not be used externally.

Quadruple-Precision Floating Point

The standard defines the encoding for the quadruple-precision floating-point data type `quadruple` (128 bits or 16 bytes). The encoding used is the IEEE standard for normalized quadruple-precision floating-point numbers [1]. The standard encodes the following three fields, which describe the quadruple-precision floating-point number.

S: The sign of the number. Values 0 and 1 represent positive and negative respectively. One bit.

E: The exponent of the number, base 2. Fifteen bits are in this field. The exponent is biased by 16383.

F: The fractional part of the number's mantissa, base 2. One hundred eleven bits are in this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

Declaration

```
quadruple identifier;
```

Quadruple-Precision Floating Point Encoding
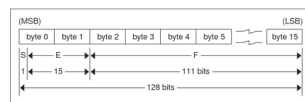
Just as the most-significant and least-significant bytes of an integer are 0 and 3, the most-significant and least-significant bits of a quadruple-precision floating-point number are 0 and 127. The beginning bit, and most-significant bit, offsets of S, E, and F are 0, 1, and 16 respectively. These offsets refer to the logical positions of the bits, not to their physical locations, which vary from medium to medium.

Consult the IEEE specifications about the encoding for signed zero, signed infinity (overflow), and de-normalized numbers (underflow) [1]. According to IEEE specifications, the `NaN` (not a number) is system dependent and should not be used externally.

Fixed-Length Opaque Data

At times, fixed-length uninterpreted data needs to be passed among systems. This data is called `opaque`.

Declaration

Opaque data is declared as follows.

```
opaque identifier[n];
```

In this declaration, the constant `n` is the static number of bytes necessary to contain the opaque data. The `n` bytes are followed by enough (0 to 3) residual zero bytes `r` to make the total byte count of the opaque object a multiple of four.

Fixed-Length Opaque Encoding

The `n` bytes are followed by enough (0 to 3) residual zero bytes `r` to make the total byte count of the opaque object a multiple of four.
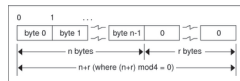
Variable-Length Opaque Data

The standard also provides for variable-length counted opaque data. Such data is defined as a sequence of `n` (numbered `0` through `n-1`) arbitrary bytes to be the number `n` encoded as an unsigned integer, as described subsequently, and followed by the `n` bytes of the sequence.

Byte `b` of the sequence always precedes byte `b+1` of the sequence, and byte `0` of the sequence always follows the sequence's length. The `n` bytes are followed by enough (0 to 3) residual zero bytes, `r`, to make the total byte count a multiple of four.

Declaration

Variable-length opaque data is declared in the following way.

```
opaque identifier<m>;
```

or

```
opaque identifier<>;;
```

The constant `m` denotes an upper bound of the number of bytes that the sequence can contain. If `m` is not specified, as in the second declaration, it is assumed to be `(2**32) - 1`, the maximum length. For example, a filing protocol might state that the maximum data transfer size is `8192` bytes, as follows.

```
opaque filedata<8192>;
```

Variable-Length Opaque Encoding

Do not encode a length greater than the maximum described in the specification.

Counted Byte Strings

The standard defines a string of `n` (numbered `0` through `n-1`) ASCII bytes to be the number `n` encoded as an unsigned integer, as described previously, and followed by the `n` bytes of the string. Byte `b` of the string always precedes byte `b+1` of the string, and byte `0` of the string always follows the string's length. The `n` bytes are followed by enough (0 to 3) residual zero bytes `r` to make the total byte count a multiple of four.

Declaration

Counted byte strings are declared as follows.

```
string object<m>;
```

or

```
string object<>;
```

The constant `m` denotes an upper bound of the number of bytes that a string can contain. If `m` is not specified, as in the second declaration, it is assumed to be `(2**32) - 1`, the maximum length. The constant `m` would normally be found in a protocol specification. For example, a filing protocol might state that a file name can be no longer than 255 bytes, as follows.

```
string filename<255>;
```

String Encoding
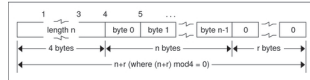
Do not encode a length greater than the maximum described in the specification.

Fixed-Length Array

Fixed-length arrays of elements numbered `0` through `n-1` are encoded by individually encoding the elements of the array in their natural order, `0` through `n-1`. Each element's size is a multiple of 4 bytes. Though all elements are of the same type, the elements might have different sizes. For example, in a fixed-length array of strings, all elements are of type `string`, yet each element varies in its length.

Declaration

Declarations for fixed-length arrays of homogeneous elements are in the following form.

```
type-name identifier[n];
```

Fixed-Length Array Encoding
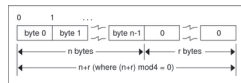
Variable-Length Array

Counted arrays enable variable-length arrays to be encoded as homogeneous elements. The element count n, an unsigned integer, is followed by each array element, starting with element `0` and progressing through element `n-1`.

Declaration

The declaration for variable-length arrays follows this form.

```
type-name identifier<m>;
```

or

```
type-name identifier<>;
```

The constant `m` specifies the maximum acceptable element count of an array. If `m` is not specified, it is assumed to be `(2**32) - 1`.

Counted Array Encoding
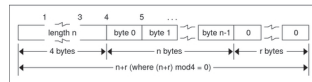
Do not encode a length greater than the maximum described in the specification.

Structure

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of 4 bytes, though the components might be different sizes.

Declaration

Structures are declared as follows.

```
struct {
   component-declaration-A;
   component-declaration-B;
   ...
 } identifier;
```

Structure Encoding

Discriminated Union

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either `int`, `unsigned int`, or an enumerated type, such as `bool`. The component types are called "arms" of the union, and are preceded by the value of the discriminant that implies their encoding.

Declaration

Discriminated unions are declared as follows.

```
union switch (discriminant-declaration) {
   case discriminant-value-A:
    arm-declaration-A;
   case discriminant-value-B:
    arm-declaration-B;
   ...
   default:
    default-declaration;
 } identifier;
```

Each `case` keyword is followed by a legal value of the discriminant. The default arm is optional. If the arm is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of 4 bytes.

The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

Discriminated Union Encoding
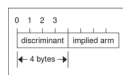
Void

An XDR `void` is a 0-byte quantity. Voids are useful for describing operations that take no data as input or no data as output. They are also useful in unions, in which some arms might contain data and others do not.

Declaration

The declaration is simply as follows.

```
void;
```

Constant

`const` is used to define a symbolic name for a constant. It does not declare any data. The symbolic constant can be used anywhere a regular constant can be used.

The following example defines a symbolic constant `DOZEN`, equal to `12`.

```
const DOZEN = 12;
```

Declaration

The declaration of a constant follows this form.

```
const name-identifier = n;
```

Typedef

`Typedef` does not declare any data either, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the `typedef`. The following example defines a new type called `eggbox` using an existing type called `egg` and the symbolic constant `DOZEN`.

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the `typedef`, if it were considered a variable. For example, the following two declarations are equivalent in declaring the variable *fresheggs*:

```
eggbox fresheggs;
 egg fresheggs[DOZEN];
```

When a `typedef` involves a `struct`, `enum`, or `union` definition, you can use another (preferred) syntax to define the same type. In general, a `typedef` of the following form:

```
typedef <<struct, union, or enum definition>> identifier;
```

can be converted to the alternative form by removing the `typedef` part and placing the identifier after the `struct`, `enum`, or `union` keyword instead of at the end. For example, here are the two ways to define the type `bool`.

```
typedef enum {/* using typedef */
   FALSE = 0,
   TRUE = 1
} bool;
enum bool {/* preferred alternative */
   FALSE = 0,
   TRUE = 1
};
```

This syntax is preferred because you do not have to go to the end of a declaration to learn the name of the new type.

Optional-Data

The optional-data union occurs so frequently that it is given a special syntax of its own for declaring it. It is declared as follows.

```
type-name *identifier;
```

This syntax is equivalent to the following union:

```
union switch (bool opted) {
  case TRUE:
  type-name element;
  case FALSE:
  void;
} identifier;
```

The optional-data syntax is also equivalent to the following variable-length array declaration, because the Boolean `opted` can be interpreted as the length of the array.

```
type-name identifier<1>;
```

Optional data is useful for describing recursive data-structures, such as linked lists and trees.

XDR Language Specification

This section contains the XDR language specification.

Notational Conventions

This specification uses a modified Backus-Naur Form notation for describing the XDR language. Here is a brief description of the notation:

- The characters |, (,), [,], and * are special.
- Terminal symbols are strings of any characters embedded in quotes (").
- Nonterminal symbols are strings of nonspecial italic characters.
- Alternative items are separated by a vertical bar (|).
- Optional items are enclosed in brackets.
- Items are grouped by enclosing them in parentheses.
- A * following an item means 0 or more occurrences of the item.

  For example, consider the following pattern:

  ```
  "a " "very" (", " " very")* [" cold " "and"] " rainy "
      ("day" | "night")
  ```

  An infinite number of strings match this pattern. A few of them are:

  ```
  a very rainy day
  a very, very rainy day
  a very cold and rainy day
  a very, very, very cold and rainy night
  ```

Lexical Notes

The following conventions are used in the specification.

- Comments begin with /* and end with */.
- White space serves to separate items and is otherwise ignored.
- An identifier is a letter followed by an optional sequence of letters, digits, or underbars (_). The case of identifiers is not ignored.
- A constant is a sequence of one or more decimal digits, optionally preceded by a minus sign (-), as seen in the following code example.

EXAMPLE 97   XDR Specification

```
Syntax Information
declaration:
 type-specifier identifier
 | type-specifier identifier "[" value "]"
 | type-specifier identifier "<" [ value ] ">"
 | "opaque" identifier "[" value "]"
 | "opaque" identifier "<" [ value ] ">"
```

```
 | "string" identifier "<" [ value ] ">"
 | type-specifier "*" identifier
 | "void"

value:
 constant
 | identifier

type-specifier:
  [ "unsigned" ] "int"
 | [ "unsigned" ] "hyper"
 | "float"
 | "double"
 | "quadruple"
 | "bool"
 | enum-type-spec
 | struct-type-spec
 | union-type-spec
 | identifier

enum-type-spec:
 "enum" enum-body

enum-body:
 "{"
 ( identifier "=" value )
 ( "," identifier "=" value )*
 "}"

struct-type-spec:
 "struct" struct-body

struct-body:
 "{"
 ( declaration ";" )
 ( declaration ";" )*
 "}"

 union-type-spec:
 "union" union-body

union-body:
 "switch" "(" declaration ")" "{"
 ( "case" value ":" declaration ";" )
 ( "case" value ":" declaration ";" )*
 [ "default" ":" declaration ";" ]
 "}"

constant-def:
 "const" identifier "=" constant ";"
```

```
type-def:
 "typedef" declaration ";"
 | "enum" identifier enum-body ";"
 | "struct" identifier struct-body ";"
 | "union" identifier union-body ";"

definition:
 type-def
 | constant-def

specification:
 definition *
```

Syntax Notes

The following are keywords and cannot be used as identifiers:

bool	float	switch
cas	hyper	typedef
chas	int	union
const	opaque	unassigned
default	quadruple	void
double	string	
enum	struct	

Only unsigned constants can be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a `const` definition.

Constant and type identifiers within the scope of a specification are in the same namespace and must be declared uniquely within this scope.

Similarly, variable names must be unique within the scope of `struct` and `union` declarations. Nested `struct` and `union` declarations create new scopes.

The discriminant of a `union` must be of a type that evaluates to an integer. That is, it must be an `int`, an `unsigned int`, a `bool`, an enum type, or any `typedef` that evaluates to one of these. Also, the `case` values must be legal discriminant values. Finally, a `case` value cannot be specified more than once within the scope of a `union` declaration.

XDR Data Description

The following example is a short XDR data description of a file data structure that might be used to transfer files from one system to another.

EXAMPLE 98   XDR File Data Structure

```
const MAXUSERNAME = 32;/* max length of a user name */
const MAXFILELEN = 65535;  /* max length of a file */
const MAXNAMELEN = 255;    /* max length of a file name */

/* Types of files: */
enum filekind {
  TEXT = 0, /* ascii data */
  DATA = 1, /* raw data */
  EXEC = 2  /* executable */
 };

/* File information, per kind of file: */
union filetype switch (filekind kind) {
  case TEXT:
   void;                         /* no extra information */
  case DATA:
   string creator<MAXNAMELEN>;     /* data creator */
  case EXEC:
   string interpreter<MAXNAMELEN>; /*proginterptr*/
};

/* A complete file: */
struct file {
  string filename<MAXNAMELEN>;        /* name of file */
  filetype type;                      /* info about file */
  string owner<MAXUSERNAME>;          /* owner of file */
  opaque data<MAXFILELEN>;            /* file data */
};
```

Suppose now that a user named linda wants to store her LISP program sillyprog that contains just the data "quit." Her file would be encoded as listed in the following table.

TABLE 20   XDR Data Description Example

Offset	Hex Bytes	ASCII	Description
0	00 00 00 09	–	Length of file name = 9
4	73 69 6c 6c	sill	File name characters
8	79 70 72 6f	ypro	More characters
12	67 00 00 00	g	3 zero-bytes of fill
16	00 00 00 02	–	Filekind is EXEC = 2
20	00 00 00 04	–	Length of interpreter = 4
24	6c 69 73 70	lisp	Interpreter characters
28	00 00 00 04	–	Length of owner = 4
32	6a 6f 68 6e	linda	Owner characters
36	00 00 00 06	–	Length of file data = 6
40	28 71 75 69	(qu	File data bytes

Offset	Hex Bytes	ASCII	Description
44	74 29 00 00	t)	2 zero-bytes of fill

RPC Language Reference

The RPC language is an extension of the XDR language. The sole extension is the addition of the program and version types.

For a description of the RPC extensions to the XDR language, see Appendix B, "RPC Protocol and Language Specification" .

D

RPC Code Examples

This appendix contains copies of the complete live code modules used in the rpcgen and RPC chapters of this book. They are compilable as they are written and will run, unless otherwise noted to be pseudo-code or the like. These examples are provided for informational purposes only. Oracle assumes no liability from their use.

Directory Listing Program and Support Routines (**rpcgen**)

EXAMPLE 99   rpcgen Program: `dir.x`

```
/*
 * dir.x: Remote directory listing
 * protocol
 *
 * This source module is a rpcgen source module
 * used to demonstrate the functions of the rpcgen
 * tool.
 *
 * It is compiled with the rpcgen -h -T switches to
 * generate both the header (.h) file and the
 * accompanying data structures.
 */
const MAXNAMELEN = 255; /*maxlengthofadirectoryentry*/
typedef string nametype<MAXNAMELEN>; /* directory entry */
typedef struct namenode *namelist; /* linkinthelisting*/

/* A node in the directory listing */
struct namenode {
    nametype name; /* name of directory entry */
    namelist next; /* next entry */
};

/*
 * The result of a READDIR operation:
 * a truly portable application would use an agreed upon list of
 * error codes rather than, as this sample program does, rely upon
```

```
 * passing UNIX errno's back. In this example the union is used to
 * discriminate between successful and unsuccessful remote calls.
 */
union readdir_res switch (int errno) {
    case 0:
        namelist list; /*no error: return directory listing*/
    default:
        void; /*error occurred: nothing else to return*/
};

/* The directory program definition */
program DIRPROG {
    version DIRVERS {
        readdir_res
        READDIR(nametype) = 1;
    } = 1;
} = 0x20000076;
```

EXAMPLE 100   Remote dir_proc.c

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>      /* Always needed */
#include <dirent.h>
#include "dir.h"          /* Created by rpcgen */

extern int errno;
extern char *malloc();
extern char *strdup();

/* ARGSUSED1*/
readdir_res *
readdir_1(dirname,req)
  nametype *dirname;
  struct svc_req *req;
{
  DIR *dirp;
  struct dirent *d;
  namelist nl;
  namelist *nlp;
  static readdir_res res; /* must be static! */

 /*
  * Open directory
  */
 dirp = opendir(*dirname);
 if (dirp == (DIR *)NULL) {
   res.errno = errno;
   return (&res);
```

```
  }
 /*
  * Free previous result
  */
 xdr_free(xdr_readdir_res, &res);
 /*
  * Collect directory entries. Memory allocated here is freed
  * by xdr_free the next time readdir_1 is called.
  */

  nlp = &res.readdir_res_u.list;
  while (d = readdir(dirp)) {
   nl = *nlp = (namenode *) malloc(sizeof(namenode));
   if (nl == (namenode *) NULL) {
    res.errno = EAGAIN;
    closedir(dirp);
    return(&res);
   }
   nl->name = strdup(d->d_name);
   nlp = &nl->next;
  }
  *nlp = (namelist)NULL;
  /* Return the result */
  res.errno = 0;
  closedir(dirp);
  return (&res);
}
```

EXAMPLE 101   rls.c Client

```
/*
 * rls.c: Remote directory listing client
 */

#include <stdio.h>
#include <rpc/rpc.h> /* always need this */
#include "dir.h"     /* generated by rpcgen */

extern int errno;

main(argc, argv)
  int argc;
  char *argv[];
{
  CLIENT *cl;
  char *server;
  char *dir;
  readdir_res *result;
  namelist nl;
```

```
 if (argc != 3) {
  fprintf(stderr, "usage: %s host directory\n",
  argv[0]);
  exit(1);
 }
 server = argv[1];
  dir = argv[2];
/*
 * Create client "handle" used for calling MESSAGEPROG on the
 * server designated on the command line.
 */
 cl = clnt_create(server, DIRPROG, DIRVERS, "visible");
  if (cl == (CLIENT *)NULL) {
   clnt_pcreateerror(server);
   exit(1);
  }

  result = readdir_1(&dir, cl);
  if (result == (readdir_res *)NULL) {
   clnt_perror(cl, server);
   exit(1);
  }

/* Okay, we successfully called the remote procedure. */
 if (result->errno != 0) {
 /*
  * A remote system error occurred. Print error message and die.
  */
 }
 if (result->errno < sys_nerr)
   fprintf (stderr, "%s : %s\n", dir,
   sys_enlist[result->errno]);
   errno = result->errno;
   perror(dir);
   exit(1);
  }

/* Successfully got a directory listing. Print it out. */
 for(nl = result->readdir_res_u.list; nl != NULL;
                        nl = nl->next) {
  printf("%s\n", nl->name);
 }
exit(0);
```

Time Server Program (rpcgen)

EXAMPLE 102   rpcgen Program: time.x

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
  version TIMEVERS {
    unsigned int TIMEGET(void) = 1;
  } = 1;
} = 0x20000044;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
% static int thetime;
%
% thetime = time(0);
% return (&thetime);
%}
#endif
```

Add Two Numbers Program (rpcgen)

EXAMPLE 103   rpcgen program: Add Two Numbers

```
/* This program contains a procedure to add 2 numbers to
demonstrate
 * some of the features of the new rpcgen. Note that add() takes 2
 * arguments in this case.
 */
program ADDPROG {   /* program number */
 version ADDVER {  /* version number */
  int add ( int, int )   /* procedure */
    = 1;
 } = 1;
} = 199;
```

Spray Packets Program (`rpcgen`)

Refer to the notes section on the man page for information about using this tool.

EXAMPLE   104   rpcgen program: spray.x

```
/*
 * Copyright (c) 1987,

2012 by

Oracle Corporation.
 */

/* from spray.x */

#ifdef RPC_HDR
#pragma ident "@(#)spray.h  1.2  91/09/17 SMI"
#endif

/*
 * Spray a server with packets
 * Useful for testing flakiness of network interfaces
 */

const SPRAYMAX = 8845; /* max amount can spray */

/*
 * GMT since 0:00, 1 January 1970
 */
struct spraytimeval {
 unsigned int sec;
 unsigned int usec;
};

/*
 * spray statistics
 */
struct spraycumul {
 unsigned int counter;
 spraytimeval clock;
};

/*
 * spray data
 */
typedef opaque sprayarr<SPRAYMAX>;

program SPRAYPROG {
```

```
version SPRAYVERS {
 /*
  * Just throw away the data and increment the counter. This
  * call never returns, so the client should always time it out.
  */
 void
 SPRAYPROC_SPRAY(sprayarr) = 1;

 /*
  * Get the value of the counter and elapsed time since last
  * CLEAR.
  */
 spraycumul
 SPRAYPROC_GET(void) = 2;

 /*
  * Clear the counter and reset the elapsed time
  */
 void
 SPRAYPROC_CLEAR(void) = 3;
 } = 1;
} = 100012;
```

Print Message Program With Remote Version

EXAMPLE 105   printmesg.c

```
/* printmsg.c: print a message on the console */
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
 char *message;

 if (argc != 2) {
  fprintf(stderr, "usage: %s <message>\n", argv[0]);
  exit(1);
 }
 message = argv[1];
 if( !printmessage(message) ) {
  fprintf(stderr, "%s: couldn't print your message\n",
         argv[0]);
  exit(1);
 }
 printf("Message Delivered!\n");
```

```
 exit(0);
}

/* Print a message to the console. */

/*
 * Return a boolean indicating whether the message was actually
 * printed.
 */
printmessage(msg)
 char *msg;
{
 FILE *f;

 if = fopen("/dev/console","w");
  if (f == (FILE *)NULL)
    return (0);
 fprintf(f,"%sen", msg);
 fclose(f);
 return (1);
}
```

EXAMPLE 106   Remote Version of `printmesg.c`

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include <rpc/rpc.h>     /* always needed */
#include "msg.h"      /* msg.h generated by rpcgen */

main(argc, argv)
 int argc;
 char *argv[];
{
 CLIENT *cl;
 int *result;
 char *server;
 char *message;
 extern int sys_nerr;
 extern char *sys_errlist[];

 if (argc != 3) {
  fprintf(stderr,"usage: %s host messagen", argv[0]);
  exit(1);
 }
 /*
  * Save values of command line arguments
  */
 server = argv[1];
```

```
 message = argv[2];
/*
 * Create client"handle" used for calling
 * MESSAGEPROG on the server
 * designated on the command line.
 */
 cl = clnt_create(server, MESSAGEPROG, PRINTMESSAGEVERS,
                  "visible");
 if (cl == (CLIENT *)NULL) {
  /*
   * Couldn't establish connection with server.
   * Print error message and die.
   */
  clnt_pcreateerror(server);
  exit(1);
 }
 /* Call the remote procedure "printmessage" on the server */
 result = printmessage_1(&message, cl);
 if (result == (int *)NULL) {
 /*
  * An error occurred while calling the server.
  * Print error message and die.
  */
  clnt_perror(cl, server);
  exit(1);
 }
 /* Okay, we successfully called the remote procedure. */
 if (*result == 0) {
  /*
   * Server was unable to print our message.
   * Print error message and die.
   */
  fprintf(stderr,"%s"
 }
/* The message got printed on the server's console */
 printf("Message delivered to %s!\n", server);
 exit(0);
}
```

EXAMPLE 107   rpcgen Program: `msg.x`

```
/* msg.x: Remote message printing protocol */
program MESSAGEPROG {
  version MESSAGEVERS {
   int PRINTMESSAGE(string) = 1;
  } = 1;
} = 0x20000001;
```

EXAMPLE 108 `mesg_proc.c`

```
/*
 * msg_proc.c: implementation of the remote
 * procedure "printmessage"
 */

#include <stdio.h>
#include <rpc/rpc.h>  /* always needed */
#include "msg.h"  /* msg.h generated by rpcgen */

/*
 * Remote version of "printmessage"
 */
/*ARGSUSED1*/
int printmessage_1(msg, req)
 char **msg;
 struct svc_req *req;
{
 static int result; /* must be static! */
 FILE *f;

 f = fopen("/dev/console", "w");
 if (f == (FILE *)NULL) {
  result = 0;
  return (&result);
 }
 fprintf(f, "%sen", *msg);
  fclose(f);
  result = 1;
  return (&result);
}
```

Batched Code Example

EXAMPLE 109   Batched Client Program

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

main(argc, argv)
 int           argc;
 char          **argv;
{
 struct timeval  total_timeout;
 register CLIENT *client;
```

```
enum clnt_stat  clnt_stat;
char            buf[1000], *s = buf;

if ((client = clnt_create(argv[1], WINDOWPROG, WINDOWVERS,
              "CIRCUIT_V")) == (CLIENT *) NULL) {
 clnt_pcreateerror("clnt_create");
 exit(1);
}

timerclear(&total_timeout);
while (scanf("%s", s) != EOF) {
 clnt_call(client, RENDERSTRING_BATCHED, xdr_wrapstring,
   &s,xdr_void, (caddr_t) NULL, total_timeout);
}

/* Now flush the pipeline */
total_timeout.tv_sec = 20;
clnt_stat = clnt_call(client, NULLPROC, xdr_void,
   (caddr_t) NULL, xdr_void, (caddr_t) NULL,
                     total_timeout);
if (clnt_stat != RPC_SUCCESS) {
 clnt_perror(client, "rpc");
 exit(1);
}
clnt_destroy(client);
exit(0);
}
```

EXAMPLE 110   Batched Server Program

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

void            windowdispatch();
main()
{
 int num;

 num = svc_create(windowdispatch, WINDOWPROG, WINDOWVERS,
       "CIRCUIT_V");
 if (num == 0) {
  fprintf(stderr, "can't create an RPC server\n");
  exit(1);
 }
 svc_run();                         /* Never returns */
 fprintf(stderr, "should never reach this point\n");
}

void
```

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

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

Non-Batched Example

This example is included for reference only. It is a version of the batched client string rendering service, written as a non-batched program.

EXAMPLE 111 Unbatched Version of Batched Client

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

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

 if ((client = clnt_create(argv[1], WINDOWPROG, WINDOWVERS,
            "CIRCUIT_V")) == (CLIENT *) NULL) {
  clnt_pcreateerror("clnt_create");
  exit(1);
 }
 total_timeout.tv_sec = 20;
 total_timeout.tv_usec = 0;
 while (scanf("%s", s) != EOF) {
  if(clnt_call(client, RENDERSTRING, xdr_wrapstring, &s,
      xdr_void, (caddr_t) NULL, total_timeout) != RPC_SUCCESS) {
   clnt_perror(client, "rpc");
   exit(1);
  }
 }
 clnt_destroy(client);
 exit(0);}
```

E

`portmap` Utility

The `rpcbind` utility replaces the `portmap` utility available in previous releases of the Oracle Solaris environment. This appendix is included to help you understand the history of port and network address resolution using the `portmap` utility.

Oracle Solaris RPC-based services use `portmap` as a system registration service. It manages a table of correspondences between ports (logical communications channels) and the services registered at them. It provides a standard way for a client to look up the TCP/IP or UDP/IP port number of an RPC program supported by the server.

System Registration Overview

For client programs to find distributed services on a network, they need a way to look up the network addresses of server programs. Network transport (protocol) services do not provide this function. Their task is to provide process-to-process message transfer across a network, that is, a message is sent to a transport-specific network address. A network address is a logical communications channel. By listening on a specific network address, a process receives messages from the network.

The way a process waits on a network address varies from one operating system to the next, but all provide mechanisms by which a process can synchronize its activity with arriving messages. Messages are not sent across networks to receiving processes, but rather to the network address at which receiving processes pick them up.

Network addresses are valuable because they allow message receivers to be specified in a way that is independent of the conventions of the receiving operating system. TI-RPC, being transport independent, makes no assumptions about the structure of a network address. It uses a universal address. This universal address is specified as a null-terminated string of characters. Such a universal address is translated into a local transport address by a routine specific to the transport provider.

The `rpcbind` protocol defines a network service that provides a standard way for clients to look up the network address of any remote program supported by a server. Because this protocol can be implemented on any transport, it provides a single solution to a general problem that works for all clients, all servers, and all networks.

`portmap` Protocol

The `portmap` program maps RPC program and version numbers to transport-specific port numbers. This program makes dynamic binding of remote programs possible.

FIGURE   11   Typical Portmap Sequence (For TCP/IP Only)
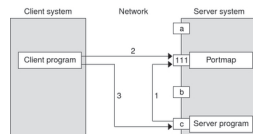
The figure illustrates the following process:

1.   The server registers with `portmap`.
2.   The client gets the server's port from `portmap`.
3.   The client calls the server.

The range of reserved port numbers is small and the number of potential remote programs is very large. By running only the port mapper on a well-known port, the port numbers of other remote programs can be ascertained by querying the port mapper. In Figure 11, "Typical Portmap Sequence (For TCP/IP Only)," on page 272 , a, 111, b, and c represent port numbers, and 111 is the assigned port-mapper port number.

The port mapper also aids in broadcast RPC. A given RPC program usually has different port number bindings on different systems, so no direct broadcasts are possible to all of these programs. The port mapper, however, does have a fixed port number. So, to broadcast to a given program, the client sends its message to the port mapper located at the broadcast address. Each port mapper that receives the broadcast then calls the local service specified by the client. When `portmap` gets the reply from the local service, it returns the reply to the client. The `portmap` protocol specification is shown in the following code example.

EXAMPLE   112   `portmap` Protocol Specification (in RPC Language)

```
const PMAP_PORT = 111;        /* portmapper port number */
 /*
  * A mapping of (program, version, protocol) to port number
  */
 struct pmap {
  rpcprog_t prog;
  rpcvers_t vers;
  rpcprot_t prot;
  rpcport_t port;
```

```
};
/*
 * Supported values for the "prot" field
 */
const IPPROTO_TCP = 6; /* protocol number for TCP/IP */
const IPPROTO_UDP = 17; /* protocol number for UDP/IP */
/*
 * A list of mappings
 */
struct pmaplist {
 pmap map;
 pmaplist *next;
};
/*
 * Arguments to callit
 */
struct call_args {
 rpcprog_t prog;
rpcvers_t vers;
rpcproc_t proc;
 opaque args<>;
};
/*
* Results of callit
*/
struct call_result {
 rpcport_t port;
 opaque res<>;
};
/*
* Port mapper procedures
*/
program PMAP_PROG {
 version PMAP_VERS {
  void
  PMAPPROC_NULL(void) = 0;
  bool
  PMAPPROC_SET(pmap) = 1;
  bool
  PMAPPROC_UNSET(pmap) = 2;
  unsigned int
  PMAPPROC_GETPORT(pmap) = 3;
  pmaplist
  PMAPPROC_DUMP(void) = 4;
  call_result
  PMAPPROC_CALLIT(call_args) = 5;
 } = 2;
} = 100000;
```

`portmap` Operation

`portmap` currently supports two protocols (UDP/IP and TCP/IP). `portmap` is contacted by talking to it on assigned port number 111 on either of these protocols. The following sections describe each of the port-mapper procedures.

PMAPPROC_NULL

This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results.

PMAPPROC_SET

When a program first becomes available on a system, it registers itself with the local port map program. The program passes its program number *prog*, version number *vers*, transport protocol number *prot*, and the port *port* on which it receives service requests. The procedure refuses to establish a mapping if one already exists for the specified *port* and it is bound. If the mapping exists and the *port* is not bound, `portmap` unregisters the *port* and performs the requested mapping. The PMAPPROC_SET procedure returns TRUE if the procedure successfully established the mapping and FALSE otherwise. See also the `pmap_set()` function in the rpc_soc(3NSL) man page.

PMAPPROC_UNSET

When a program becomes unavailable, it should unregister itself with the port-mapper program on the same system. The parameters and results of PMAPPROC_UNSET have meanings identical to those of PMAPPROC_SET. The *protocol* and *port number* fields of the argument are ignored. See also the `pmap_unset()` function in the rpc_soc(3NSL) man page.

PMAPPROC_GETPORT

Given a program number *prog*, version number *vers*, and transport protocol number *prot*, the PMAPPROC_GETPORT procedure returns the port number on which the program is awaiting call requests. A port value of zeroes means the program has not been registered. The *port* field of

the argument is ignored. See also the `pmap_getport()` function in the rpc_soc(3NSL) man page.

PMAPPROC_DUMP

The `PMAPPROC_DUMP` procedure enumerates all entries in the port mapper's database. The procedure takes no parameters and returns a list of `program`, `version`, `protocol`, and `port` values. See also the `pmap_getmaps()` function in the rpc_soc(3NSL) man page.

PMAPPROC_CALLIT

The `PMAPPROC_CALLIT` procedure enables a caller to call another remote procedure on the same system without knowing the remote procedure's port number. It supports broadcasts to arbitrary remote programs by using the well-known port mapper's port. The parameters *prog, vers, proc*, and the bytes of *args* are the program number, version number, procedure number, and parameters of the remote procedure. See also the `pmap_rmtcall()` function in the rpc_soc(3NSL) man page.

This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise. It also returns the remote program's `port number`, and the bytes of `results` are the results of the remote procedure.

The port mapper communicates with the remote program using UDP/IP only.

Glossary

RPC Programming Terms

client A process that remotely accesses resources of a computer server

client handle A client process data structure that represents the binding of the client to a particular server's RPC program.

connection-oriented transport Characteristic of the model of interconnection in which communication proceeds through three well-defined phases: connection establishment, data transfer, and connection release. See *stream transport*.

connectionless transport Characteristic of the model of interconnection in which communication takes place without first establishing a connection. See *datagram transport*.

datagram transport A message and the Internet source and destination addresses that are associated with it. Datagram transports have less overhead than connection-oriented transports but are considered less reliable. Data transmissions are limited by buffer size.

deserialize To convert data from XDR format to a system-specific representation.

handle An abstraction used by the service libraries to refer to a file or a file-like object such as a socket.

host A computer system that is accessed by computers and/or workstations at remote locations. Usually the host contains the data, but in networks, the remote locations can be the host and provide information to the network.

MT hot Characteristic of an interface in which the library or call automatically creates threads.

MT safe Characteristic of an interface that can be called in a threaded environment. An MT-safe interface can be invoked concurrently for multiple threads.

network client A process that makes remote procedure calls to services.

network server A network device that manages resources and supplies services to a client.

network service A collection of one or more remote service programs.

ping	A service that verifies activity on a remote system. A computer sends a small program to a host and notes time on its return path.
remote program	A program that implements one or more remote procedures.
RPC language (RPCL)	A C-like programming language translated by the `rpcgen` compiler. RPCL is a superset of XDR Language.
RPC library	The network services library, libnsl, specified to the link editor at compile time. Also known as the RPC package.
RPC protocol	The message-passing protocol that is the basis of the RPC package.
RPC/XDR	A standard for system-independent data structures. See *RPC language*.
serialize	To convert data from a system representation to XDR format.
server	A network device that manages resources and supplies services to a client.
TI-RPC	Transport-independent RPC. This is the current version.
transport	The fourth layer of the Open Systems Interconnection (OSI) Reference Model.
transport handle	An abstraction used by the RPC libraries to refer to the transport's data structures.
TS-RPC	Transport-specific RPC. This is an older version. It is still supported, but TI-RPC is preferred.
universal address	A hexadecimal address of a type of network, such as TCP/IP, that configures the port monitor to check for print requests from print clients on a network.
virtual circuit transport	An apparent connection between processes that is facilitated by transmission control protocol (TCP). A virtual circuit enables applications to "talk" to each other as if they had a physical circuit.
XDR Language	A data description language and data representation protocol.

Index