R199-02

VOS Communications Software: Remote Procedure Call (RPC) Facility

Stratus Technologies

Notice

The information contained in this document is subject to change without notice.

UNLESS EXPRESSLY SET FORTH IN A WRITTEN AGREEMENT SIGNED BY AN AUTHORIZED REPRESENTATIVE OF STRATUS TECHNOLOGIES, STRATUS MAKES NO WARRANTY OR REPRESENTATION OF ANY KIND WITH RESPECT TO THE INFORMATION CONTAINED HEREIN, INCLUDING WARRANTY OF MERCHANTABILITY AND FITNESS FOR A PURPOSE. Stratus Technologies assumes no responsibility or obligation of any kind for any errors contained herein or in connection with the furnishing, performance, or use of this document.

Software described in Stratus documents (a) is the property of Stratus Technologies Bermuda, Ltd. or the third party, (b) is furnished only under license, and (c) may be copied or used only as expressly permitted under the terms of the license.

Stratus documentation describes all supported features of the user interfaces and the application programming interfaces (API) developed by Stratus. Any undocumented features of these interfaces are intended solely for use by Stratus personnel and are subject to change without warning.

Manual Name: *VOS Communications Software: Remote Procedure Call (RPC) Facility*

Part Number: R199
Revision Number: 02
OpenVOS Release Number: 17.2.0
Printing Date: March 2013

Stratus Technologies, Inc.
111 Powdermill Road
Maynard, Massachusetts 01754-3409

Preface

The Purpose of This Manual

The manual *VOS Communications Software: Remote Procedure Call (RPC) Facility* (R199) documents the Remote Procedure Call (RPC) facility.

Audience

This manual is intended for RPC administrators and programmers. Users, administrators, and programmers should use this manual as follows:

- Programmers should read the overview of RPC in Chapter 1.
- Programmers who want to use RPC to communicate between a workstation or system running the UNIX® operating system and a Stratus system running the VOS operating system should be familiar with the information in Chapters 3 through 5.

Before using the manual *VOS Communications Software: Remote Procedure Call (RPC) Facility* (R199), you should be familiar with the following manuals.

- *VOS Communications Software: Multiplexed Generic Communications Software* (R095)
- *OpenVOS System Administration: Configuring a System* (R287)
- The appropriate workstation manuals

You should also be familiar with the user's guide for one of the following two Stratus TCP/IP products that your system is running.

- The **TCP/IP Version 2** product consists of firmware that is loaded onto the K104 Ethernet Communications I/O Adapter. For information about how to administer this product, see the manual *VOS Communications Software: TCP/IP Administration* (R196).
- The **OS TCP/IP** product is the Stratus implementation of the Berkeley Software Distribution (BSD) UNIX Version 4.3 of TCP/IP. This product provides a protocol driver that is bound with the OpenVOS kernel. For information about how to administer this product, see the *VOS OS TCP/IP Administrator's Guide* (R223).

Revision Information

This manual is a revision. For information on which release of the software this manual documents, see the Notice page. Change bars, which appear in the margin, note the specific changes to text. In this revision, information about the NFS facility was removed because NFS is no longer supported.

Manual Organization

This manual has five chapters and one appendix. The book discusses the RPC facility, including programming with RPC and External Data Representation (XDR) function calls.

Chapter 1, "Overview of the RPC Facility," provides an overview of the Remote Procedure Call (RPC) facility and discusses RPC terminology, how XDR is used by RPC, and the client and server sides of RPC.

Chapter 2, "Administering the RPC Facility," describes how to configure and monitor RPC, how to start and stop RPC, and how to control security when using RPC.

Chapter 3, "RPC Programming," provides specific information about programming with RPC.

Chapter 4, "RPC and XDR Function Calls," provides a detailed description of each of the RPC and XDR function calls.

Chapter 5, "Compiling, Binding, and Debugging RPC Programs," discusses the include files and object modules necessary for compiling and binding RPC programs, and provides information on the OpenVOS debugger.

Appendix A, "Sample Programs," presents a client program and a server program that use RPC and XDR function calls.

Notation

Stratus documentation uses *italics* to introduce or define new terms. For example:

The *mount point* is your point of access to the remote file system.

`Computer font` represents text that would appear on your display screen or on a printer. For example:

When logged in as a superuser, issue the `mount` command.

`Slanted font` represents general terms that are to be replaced by literal values. In the following example, the user must replace the slanted-font term with an actual value.

```
mount "server_name:server_path_name" mount_point
```

Boldface emphasizes words within the text. For example:

Every module **must** have a copy of the `module_start_up.cm` file.

Related Manuals

Refer to the following Stratus manuals for related documentation.

- TCP/IP Version 2 manuals:

  VOS Communications Software: TCP/IP User's Guide (R197)
  VOS Communications Software: TCP/IP Administration (R196)
  VOS Communications Software: TCP/IP Programmer's Guide (R129)

- *VOS Communications Software: Multiplexed Generic Communications Software (R095)*

- *Installing a Major or Update Release of a Product* (R217)

- *VOS Ethernet Protocol Support* (R128)

- *VOS C Language Manual* (R040)

- VOS System Administration manuals:

  VOS System Administration: Administering and Customizing a System (R281)
  VOS System Administration: Starting Up and Shutting Down a Module or System (R282)
  OpenVOS System Administration: Registration and Security (R283)
  OpenVOS System Administration: Disk and Tape Administration (R284)
  OpenVOS System Administration: Backing Up and Restoring Data (R285)
  VOS System Administration: Administering the Spooler Facility (R286)
  OpenVOS System Administration: Configuring a System (R287)

Online Documentation

You can find additional information by viewing the system's online documentation in `>system>doc`. The online documentation contains the latest information available, including updates and corrections to Stratus manuals.

A Note on the Contents of Stratus Manuals

Stratus manuals document all of the subroutines and commands of the user interface. Any other commands and subroutines contained in the operating system are intended solely for use by Stratus personnel and are subject to change without warning.

How to Comment on This Manual

You can comment on this manual by using the command `comment_on_manual`, which is documented in the manual *OpenVOS System Administration: Administering and Customizing a System* (R281) and the *OpenVOS Commands Reference Manual* (R098). There are two ways you can use this command to send your comments.

- If your comments are brief, type `comment_on_manual`, press ENTER or RETURN, and complete the data-entry form that appears on your screen. When you have completed the form, press ENTER.

- If your comments are lengthy, save them in a file before you issue the command. Type `comment_on_manual` and press DISPLAY_FORM. Enter this manual's part number, `R199`, and then enter the name of your comments file in the `-comments_path` field. Press CYCLE to change the value of `-use_form` to `no` and then press ENTER.

Your comments are sent to Stratus over the Remote Service Network. Note that the operating system includes your name with your comments.

Stratus welcomes any corrections and suggestions for improving this manual.

Contents

Tables

Figures

Chapter 1:
Overview of the RPC Facility

This chapter provides an overview of the Remote Procedure Call (RPC) facility. It discusses RPC terminology, how RPC uses XDR calls, and the client and server sides of RPC.

RPC enables communication with servers in a manner similar to the procedure-calling mechanism available in many programming languages. The RPC protocol consists of a library of function calls and a specification for portable data transmission, known as External Data Representation (XDR). Both RPC and XDR are portable, providing a standard I/O library for interprocess communication. The RPC and XDR function calls provide programmers with a standardized method for accessing sockets; programmers need not be concerned with the low-level details of the TCP/IP `accept()`, `bind()`, and `connect()` functions. Using RPC and XDR function calls, you can write C programs that enable communication between Stratus modules and UNIX-based modules. In the Stratus implementation of RPC, the Stratus module can be both a client and a server.

RPC Terminology

For RPC, a *server* is a collection of one or more remote programs. A server may support more than one *version* of a remote program in order to be compatible with various versions of programs. A *remote program* contains one or more *remote procedures*. Each remote procedure implements a *service*. A *client* is a program that makes remote procedure calls to servers. In RPC architecture, clients send a *call message* and wait for servers to return a *reply message*.

RPC provides a mechanism that enables a client process to have the server process execute a procedure call as if the client process had executed the procedure call in its own address space. Because the client and the server are two separate processes, they do not have to exist on the same physical machine.

The *Portmap server* associates RPC program numbers with IP port numbers. As part of calling a server, a client typically needs to access a remote program on the server at the port with which the remote program is associated. Clients use the Portmap server to look up the port numbers of remote programs. Once the clients have obtained the port number, they send requests directly to the remote program's port.

XDR

XDR function calls are used to transmit data that is accessed by more than one type of machine. The XDR function calls solve data portability problems; they enable you to read and write arbitrary C constructions in a consistent manner. The function calls ensure consistency even when the data is shared among different machines on the same network.

The XDR function calls include filter routines for transmitting data types, such as strings (null-terminated arrays of bytes), structures, unions, and arrays. Using the basic built-in function calls, you can write XDR routines to describe arbitrary data structures, such as elements of arrays, arms of unions, or objects pointed to from other structures. The structures themselves may contain arrays of arbitrary elements or pointers to other structures. See Chapter 3, ‘‘RPC Programming” for more information on user-defined function calls.

The Client Side of RPC

On the client side of RPC, the client sends a call message to a remote program requesting a particular service on the server. The first time a client calls a remote program, RPC automatically routes the call through the Portmap server. The Portmap server returns to the client the appropriate port number in a reply message. The client then sends call messages to the remote program’s port. When a call message arrives, the server calls the appropriate procedure, which performs the requested service. The server returns the reply message and the procedure call returns to the client. See Chapter 3, ‘‘RPC Programming” for more information on the client side of the RPC program.

The Server Side of RPC

On the server side of RPC, a server is a collection of one or more programs, and a program contains one or more procedures. A server process registers programs and procedures with the Portmap server. The server process remains dormant while it waits for a call message from a client. When a call message arrives, the server process responds to the call by extracting parameters from the appropriate procedure’s parameters. The server process computes the results, sends a reply message to the client, and then waits for the next call message. Only one of the two processes (client or server) is active at any given time; that is, the RPC protocol does not support multithreading (the concurrent execution of multiple tasks) of client and server processes. See Chapter 3, ‘‘RPC Programming” for more information on the server side of the RPC program.

Chapter 2:
Administering the RPC Facility

This chapter explains how to install, configure, and monitor RPC. It also describes how to start and stop RPC and how to control security in an RPC program.

Installing and Configuring RPC

As part of installing and configuring RPC, you must edit the RPC programs database file (`rpc_programs.db`) so that it includes every RPC program that will run on your system and on any remote systems. This section describes the RPC programs database file.

The RPC Programs Database File

The database file `rpc_programs.db` maps RPC program names to their corresponding program numbers. The `rpc_programs.db` file, which is a standard ASCII file, must be located in the directory `>system>tcp` if you are using TCP/IP Version 2. The `rpc_programs.db` file is a table, the first line of which contains the names of the fields separated by colons (`:`). This first line, which you should **not** modify, must appear exactly as shown here.

```
programname:number:comments
```

The first line must be left-justified, with no leading or trailing blanks. All other lines in the database file can contain leading or trailing blanks.

The `rpc_programs.db` file contains three fields.

- `programname` specifies the name of the RPC program.
- `number` specifies the number of the RPC program.
- `comments` provides specific information about the program names.

An example of an `rpc_programs.db` file follows.

```
programname:number:comments
portmap:    100000:   The program number for the Portmap server
nfs:        100003:   The program number for the Network File System
mountd:     100005:   NFS server's mount daemon
ticker:   30111200:   The program number for the Ticker Plant
```

The RPC Programs Database File at Installation

A sample RPC programs database file, `rpc_programs.db.base`, is located in the directory `>system>rpc_command_library`. If your module is running TCP/IP Version 2, copy this

file to the directory `>system>tcp`; if your module is running OS TCP/IP, copy this file to the directory `>system>tcp_os`. Then, for both TCP/IP products, rename the file `rpc_programs.db`, and add to it the names of the RPC programs you will run on your system. For information on modifying the file, see ''Modifying the RPC Programs Database File'' on page 2-2. For information on assigning program numbers, see ''Assigning Program Numbers'' on page 3-14.

> **Caution: Subsequent installations of TCP/IP Version 2 will delete the** `rpc_programs.db` file in the directory `>system>tcp`; subsequent installations of OS TCP/IP will delete the `rpc_programs.db` file in the directory `>system>tcp_os`. Be sure to back up this file so that you can restore it after any TCP/IP installations.

Modifying the RPC Programs Database File

To modify the `rpc_programs.db` file, copy the `rpc_programs.db.base` file in the directory `>system>rpc_command_library` to the appropriate directory (`>system>tcp` or `>system>tcp_os`); then, rename the file `rpc_programs.db`. Or, you can simply modify the `rpc_programs.db` file that already resides in the `>system>tcp` or `>system>tcp_os` directory. Edit the `rpc_programs.db` file, adding any RPC programs that you will run on your system. This file must remain in the `>system>tcp` or `>system>tcp_os` directory. The following steps present one method for modifying the RPC programs database file.

1. Change to the directory `>system>rpc_command_library` by issuing the following command.

   ```
   change_current_dir
   (master_disk)>system>rpc_command_library
   ```

2. Copy the `rpc_programs.db.base` sample file to the `rpc_programs.db` file in the appropriate directory (`>system>tcp` or `>system>tcp_os`) by issuing one of the following commands.

   ```
   copy_file rpc_programs.db.base >system>tcp>rpc_programs.db

   copy_file rpc_programs.db.base
   >system>tcp_os>rpc_programs.db
   ```

3. Change from your current directory to the appropriate directory (`>system>tcp` or `>system>tcp_os`) by issuing one of the following commands.

   ```
   change_current_dir >system>tcp

   change_current_dir >system>tcp_os
   ```

4. Use a OpenVOS editor to update and save the `rpc_programs.db` file.

   Caution: Subsequent installations of TCP/IP will delete the `rpc_programs.db` file in the directory `>system>tcp` or `>system>tcp_os`.

Starting and Stopping RPC

To run the Portmap server on a K104 Ethernet Communications I/O Adapter other than the default, specify a value for the optional *adapter* argument when issuing the `portmap` command. The *adapter* argument also allows you to specify the OS TCP/IP protocol driver for a module running OS TCP/IP. For example, the following command starts the Portmap server on the nondefault adapter `%es#enet.24.2` on a module running TCP/IP Version 2.

```
start_process '>system>rpc_command_library>portmap %es#enet.24.2' &+
     -process_name portmap -output_path &+
     >system>rpc_command_library>portmap.out -privileged &+
     -priority 8 -current_dir >system>rpc_command_library
```

When writing an RPC program, you specify an adapter other than the default by calling the `s$tcp_set_default_adapter()` function in the program. For information on the `s$tcp_set_default_adapter()` function call, see the *VOS Communications Software: TCP/IP Programmer's Guide* (R129) for TCP/IP Version 2.

If the Portmap server is stopped for any reason, remember that **all** RPC server programs must be stopped as well. RPC server programs include all servers (including any created at your site) that use RPC.

To stop the Portmap server, issue the following command.

```
stop_process portmap
```

Monitoring RPC

The `rpcinfo` command displays information about the RPC services running on an RPC host, including the program numbers, version numbers, protocols, ports, and service names. You can issue the `rpcinfo` command on any RPC host. The RPC host, specified by the *host_name* argument of the `rpcinfo` command, is any computer system that resides on a TCP/IP network and runs the Portmap server.

This section explains how to use the `-p`, `-u`, and `-t` arguments of the `rpcinfo` command.

Listing the RPC Programs

To list the RPC programs (or services) that are registered with the Portmap server, issue the `rpcinfo` command with the `-p` argument to specify the name of the host on which the Portmap server is running. For example, the following command displays a list of the RPC services available on the RPC host `stratus`.

```
rpcinfo -p stratus
```

The command output follows, listing the program numbers, version numbers, protocols, ports, and service names.

```
program  vers  proto    port
 100000     2    tcp    111    portmapper
 100000     2    udp    111    portmapper
 100003     2    udp    2049   nfs
 100005     1    udp   11738   mountd
```

As another example, the following command displays a list of the RPC services available on the RPC host `sun`.

```
rpcinfo -p sun
```

The command output follows, listing the program numbers, version numbers, protocols, ports, and service names.

```
program  vers  proto    port
 100024     1    udp    1026   status
 100024     1    tcp    1024   status
 100021     1    tcp    1025   nlockmgr
 100021     1    udp    1031   nlockmgr
 100020     1    udp    1034   llockmgr
 100003     2    udp    2049   nfs
 100020     1    tcp    1026   llockmgr
 100021     2    tcp    1027   nlockmgr
 100012     1    udp    1057   sprayd
 100011     1    udp    1059   rquotad
 100005     1    udp    1061   mountd
 100008     1    udp    1063   walld
 100002     1    udp    1065   rusersd
 100002     2    udp    1065   rusersd
 100001     1    udp    1068   rstatd
 100001     2    udp    1068   rstatd
 100001     3    udp    1068   rstatd
 100015     6    udp    1283   selection_svc
```

Listing UDP Protocol Services

To determine whether a program that is using UDP is responding to requests, issue the `rpcinfo` command with the `-u` argument to specify the program name or number and the `-p` argument to specify the name of the host on which the program is running. This command sends an RPC call to the specified program on the specified host. For example, the following command sends an RPC call to the program `nfs`, which is using UDP on the RPC host `stratus`.

```
rpcinfo -p stratus -u nfs
```

This command results in the following message, which indicates that version 2 of the program is responding to requests.

```
program 100003 version 2 ready and waiting
```

The `rpcinfo` command also displays responses from multiple versions of a program, if those versions are registered with the Portmap server. For example, the following command sends an RPC call to program number `100001`, which is using UDP on the RPC host `stratus`.

```
rpcinfo -p stratus -u 100001
```

This command results in the following message, which indicates that three versions of the program are responding to requests.

```
program 100001 version 1 ready and waiting
program 100001 version 2 ready and waiting
program 100001 version 3 ready and waiting
```

Listing TCP Protocol Services

To determine whether a program that is using the TCP transport protocol is responding to requests, issue the `rpcinfo` command with the `-t` argument to specify the program name or number and the `-p` argument to specify the name of the host on which the program is running. This command sends an RPC call to the specified program on the specified host. For example, the following command sends an RPC call to program number `100020`, which is using the TCP transport protocol on the RPC host `stratus`.

```
rpcinfo -p stratus -t 100020
```

This command results in the following message, which indicates that the program is responding to requests.

```
program 100020 version 1 ready and waiting
```

Controlling Security in an RPC Program

You can control security in an RPC program by using any of the following three authentication function calls.

```
authnone_create()
authunix_create()
auth_destroy()
```

The authentication function calls maintain the fields necessary for clients to identify themselves to a server. These fields are referred to as *credentials*. For more information on the fields of the authentication function calls, see ‘‘Incorporating Authentication’’ on page 3-21.

You can specify different types of authentication (`AUTH_UNIX` or `AUTH_NULL`) for call and reply messages just as you can use different transport protocols (TCP or UDP). The `AUTH_UNIX` authentication type allows a caller of a remote procedure to identify itself using the same credentials (such as `UID` and `GID`) that a user on a UNIX system uses for identification. The `AUTH_NULL` authentication type is the default if you specify no authentication type.

A client program uses the authentication function calls to create the desired credentials and then to destroy the credentials when they are no longer needed. The `authnone_create()`

function call creates credentials with the `AUTH_NULL` authentication type, and the `authunix_create()` function call creates credentials with the `AUTH_UNIX` authentication type. The `auth_destroy()` function call destroys the credentials of both authentication types. These credentials are passed in the call message to the server program, which determines whether the call message will be accepted or rejected. Thus, each client program must implement its own type of authentication and the server program must reflect this type in the return values of its reply messages. For more detailed information on RPC authentication, see "Incorporating Authentication" on page 3-21.

Chapter 3:
RPC Programming

This chapter discusses RPC programming, the general structure of the RPC client side and server side of a program, assigning program numbers, using XDR function calls to pass data structures and allocate memory, and incorporating authentication into RPC programs. It also discusses RPC tasks such as batching and broadcasting.

You must be familiar with the following special procedures to ensure a successful interface between RPC programs and the version of TCP/IP software your module is running.

- If your module is running OS TCP/IP, you **must** bind RPC programs with the object modules located in the directory `(master_disk)>system>rpc_tcp_os_object_library` rather than the default directory `(master_disk)>system>rpc_object_library`. You should ensure that your object-library search paths include this directory. (If your module is running TCP/IP Version 2, you bind RPC programs with the object modules located in the directory `(master_disk)>system>rpc_object_library`, which is the default object-library search path.)

- Any program that uses RPC function calls **must** contain an `#include` statement for the include file `tcp_socket.h`, which contains TCP/IP and socket definitions. The statement `#include <tcp_socket.h>` must appear in the program **before** `#include` statements for any RPC include files. For TCP/IP Version 2, the file `tcp_socket.h` is located in the directory `(master_disk)>system>tcp/ip_include_library`. For OS TCP/IP, this file is located in the directory `(master_disk)>system>tcp_os>include_library`. You should ensure that the appropriate include library is in the list of include-library search paths.

  Caution: If you intend to convert the module on which you execute RPC programs from TCP/IP Version 2 to OS TCP/IP, you must be familiar with the information in the file `os_tcp_port_guide.doc`, which is located in the directory `(master_disk)>system>doc>tcp_os`. This file, entitled ***Porting Guide: TCP/IP 2.0 to OS TCP/IP***, explains how to port programs from TCP/IP Version 2 to OS TCP/IP.

RPC Programming Capabilities

RPC enables a client to send a call message to a procedure on a remote server. The server then calls the procedure, which executes the request, and sends a reply message to the client. RPC performs the following tasks:

- specifies a procedure to be called
- matches reply messages to call messages
- checks authentication of the client and the server.

RPC also detects the following error conditions:

- RPC protocol version mismatches
- remote procedure version mismatches
- protocol errors, such as failure to pass the correct arguments to a procedure
- remote authentication failure
- failure to call the specified remote procedure.

Each RPC procedure is defined by a program number, program version number, and procedure number, as shown in Figure 3-1. Thus, to uniquely identify an RPC procedure, an RPC call message contains three unsigned fields: program number (`prognum`), program version number (`versnum`), and procedure number (`procnum`). The program number defines a group of related remote procedures, each having a different procedure number. Each program also has a version number, so if you make a minor change (such as adding a new procedure), you need not assign a new program number. The subsection "Assigning Program Numbers" later in this chapter presents more information on program numbers. The subsection "Supporting Multiple Versions of a Program" later in this chapter presents more information on supporting multiple program versions within a remote procedure.

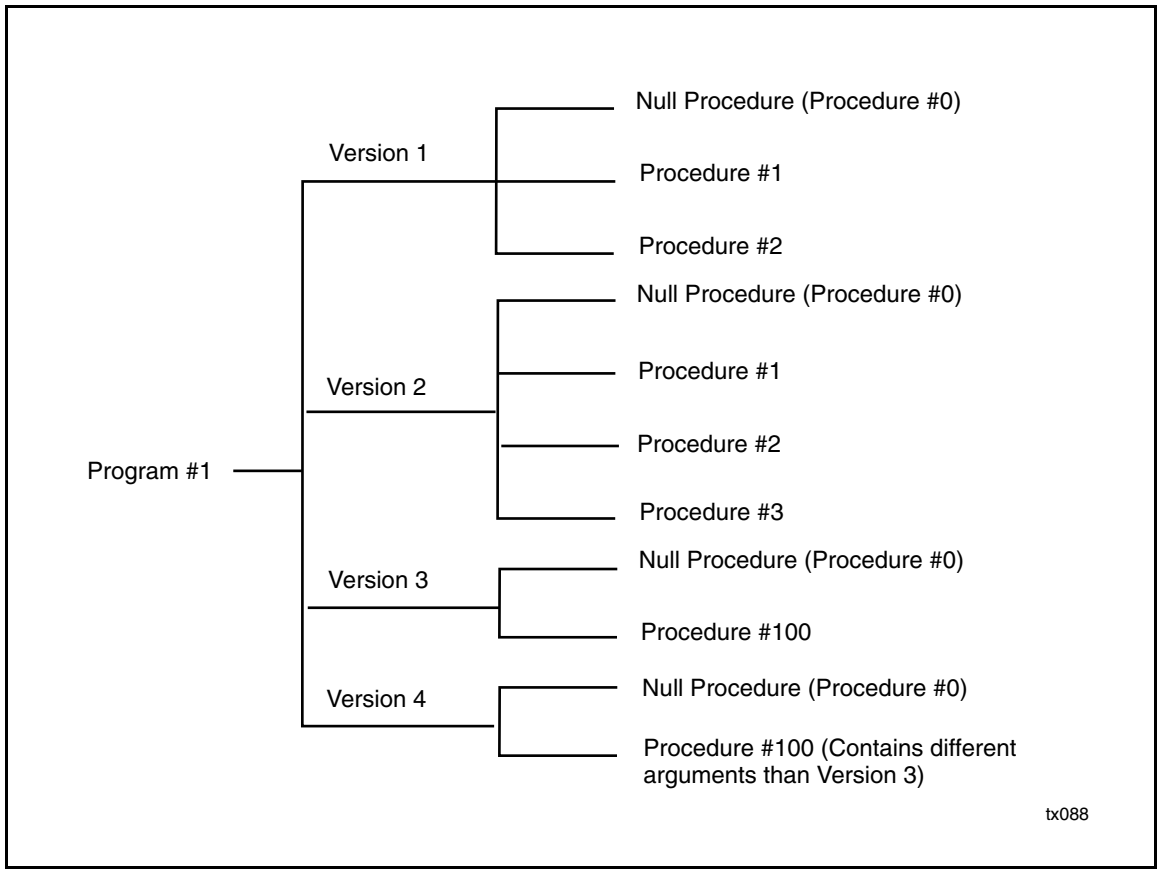

Figure 3-1. Relationship of RPC Programs, Versions, and Procedures

Just as programs may change over several versions, the actual RPC message protocol could also change. Therefore, the call message also has the RPC version number in it; the value of this field is always `2`.

The reply message to a call message passes back any requested data and also contains appropriate error messages for any errors that have occurred. Error conditions that could be included in the reply message are listed below.

- The remote implementation of RPC is not using RPC protocol version 2. (The lowest and highest supported RPC version numbers are returned as part of this message.)
- The remote program is not available on the remote system.
- The remote program does not support the requested program version number. (The lowest and highest supported remote program version numbers are returned as part of this message.)
- The requested procedure number does not exist. (This is usually a client-side protocol or programming error.)
- The arguments to the remote procedure appear to be incorrect from the server's point of view. (This is caused by an inconsistency in the protocol between client and server.)

The function calls that handle these errors are discussed in the section "Server Error Function Calls" later in this chapter.

The General Structure of an RPC Program

A server registers its procedures with the Portmap server and then waits for client call messages. The first time a client sends a call message to a remote procedure, RPC automatically routes the call through the Portmap server, which then directs the call to the appropriate server. The server sends the client a reply message, which includes the server's port address. After the client receives the port address, the client can call the server directly. When the server receives a client call message, the server calls the appropriate procedure, which performs the requested service and returns a reply to the server. The server then sends a reply message to the client. Figure 3-2 illustrates this process.

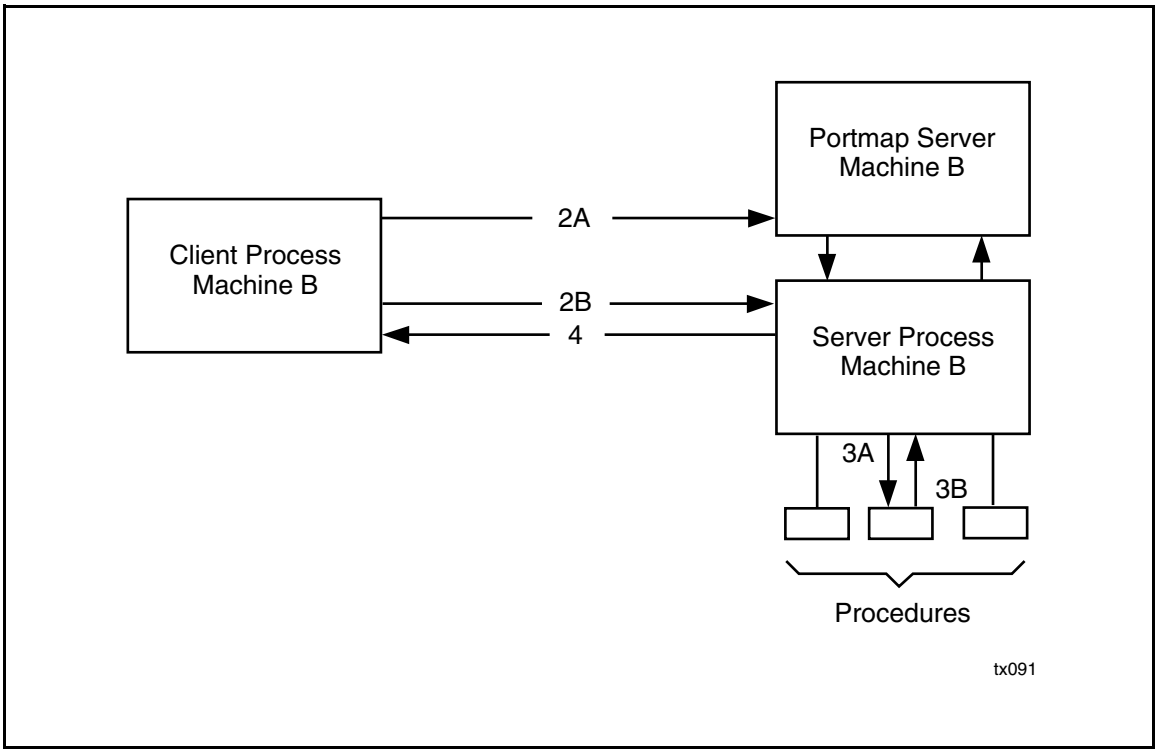

Figure 3-2. RPC Calls and the Portmap Server

The steps in Figure 3-2 are as follows:

Step1. The server registers its procedures with the Portmap server and then waits for a call message.

Step 2A. The client sends a call message to the server. RPC routes the call to the Portmap server, which directs the call to the appropriate server. The server sends the client a reply message, which includes the server's port address.

Step 2B. The client then calls the server directly.

Step 3A. The server calls the appropriate procedure.

Step 3B. The procedure performs the requested service and sends a reply message to the server.

Step4. The server sends a reply message to the client.

The `registerrpc()` function call registers a C procedure within the server so that it corresponds with an RPC procedure number within the Portmap server. If you have more than one procedure, call `registerrpc()` once for each procedure. You may want to use the `svc_register()` function call instead of `registerrpc()` to register procedures, since `svc_register()` registers only the program and version numbers, whereas `registerrpc()` registers program, version, and procedure numbers. This feature of `svc_register()` allows registration of a dispatch routine.

A *dispatch routine* is a routine that determines which procedure to call, based on procedure and version numbers. Thus, instead of registering procedures individually, a number of procedures within a dispatch routine may be registered with one call. Additional reasons for using `svc_register()` instead of `registerrpc()` are discussed later in this section.

Choosing the Transport Protocol

RPC procedure calls can be made using either the UDP or the TCP transport protocol. UDP is an *unreliable* protocol, meaning it will not wait for a response from a server when a data packet is sent to the server. UDP restricts RPC calls to 4050 bytes of data. When using UDP with RPC, the error recovery of lost messages is handled by RPC with the use of time-outs to initiate retransmission. When using UDP and waiting on reply messages, programmers usually set a time-out in the client process for the length of time to wait for reply messages.

Although the length of time-outs is program dependent, the length of a short time-out is generally four times the length of the server response time, and the length of a long time-out is greater than four times the server response time.

For example, if the server response time is 2 seconds, a short time-out would be 8 seconds. If the client retransmits call messages after short time-outs, the absence of a reply message can mean either that the remote procedure was not executed or it was executed an unknown number of times. It can be inferred from the receipt of a reply message that the remote procedure was executed at least once.

Most RPC function calls use UDP, but in some cases programs must send long streams of data and therefore may use TCP. TCP is a *reliable* protocol, meaning it will wait for a response from the server before sending it more data. The absence of a reply message means that the remote procedure was not executed or was executed once and the reply message was lost. Receipt of a reply message means that the remote procedure was executed once. The TCP protocol is approximately half as fast as the UDP protocol.

Choosing the RPC Function Calls

The `callrpc()` function call on the client side and the `registerrpc()` function call on the server side take care of the necessary steps for making remote procedure calls and are the simplest function calls to use. The `callrpc()` function call automatically chooses the UDP transport protocol, sets up the client transport handle, and issues the call message. The `registerrpc()` function call uses the UDP transport protocol and registers only one procedure at a time.

There are times when you should use function calls other than `callrpc()` and `registerrpc()`.

- You may need to send long streams of data using the TCP transport protocol, which allows streams of data larger than those permitted by the UDP transport protocol.
- You may need to allocate and deallocate memory while serializing or deserializing data with XDR function calls. For a detailed explanation of memory allocation, see “Allocating Memory” on page 3-20.

- You may want to authenticate call messages by supplying credentials and verifying them. For a detailed explanation of authentication, see "Incorporating Authentication" on page 3-21.
- You may want to alter the default values of the function calls.
- You may want to specify a socket. The `callrpc()` function call passes the `RPC_ANYSOCK` field. For more information on selecting sockets, see the *VOS Communications Software: TCP/IP Programmer's Guide* (R129) for TCP/IP Version 2.

The sections "The General Structure of the RPC Client Side" and "The General Structure of the RPC Server Side," later in this chapter, first discuss the use of the simple RPC function calls and then show how more elemental RPC function calls may be used to implement the tasks listed above.

Selecting Sockets

You cannot specify the socket to be used when `callrpc()` creates a client transport handle or when `registerrpc()` creates a server transport handle. The `callrpc()` and `registerrpc()` function calls use `RPC_ANYSOCK` when creating transport handles. The `RPC_ANYSOCK` field causes a socket to be selected automatically for the creation of a transport handle.

To have control over the socket used in the client transport handle, you must use the `clntudp_create()` or `clnttcp_create()` function call. Similarly, to have control over the socket used in the server transport handle, you must use the `svcudp_create()` or `svctcp_create()` function call. To have the socket selected automatically for you, you can specify `RPC_ANYSOCK` as the socket argument for these transport handle creation routines. Or, you may specify the address of the socket as the socket argument for the transport handle creation routines.

For detailed information about the transport handle creation routines, see "Using Function Calls Instead of callrpc()" on page 3-9 and "Using Function Calls Instead of registerrpc()" on page 3-12. For more information on selecting sockets, see the *VOS Communications Software: TCP/IP Programmer's Guide* (R129) for TCP/IP Version 2.

The General Structure of the RPC Client Side

This section describes the function calls used on the RPC client side of a program. Figure 3-3 presents a flowchart of the calls typically used on the RPC client side.

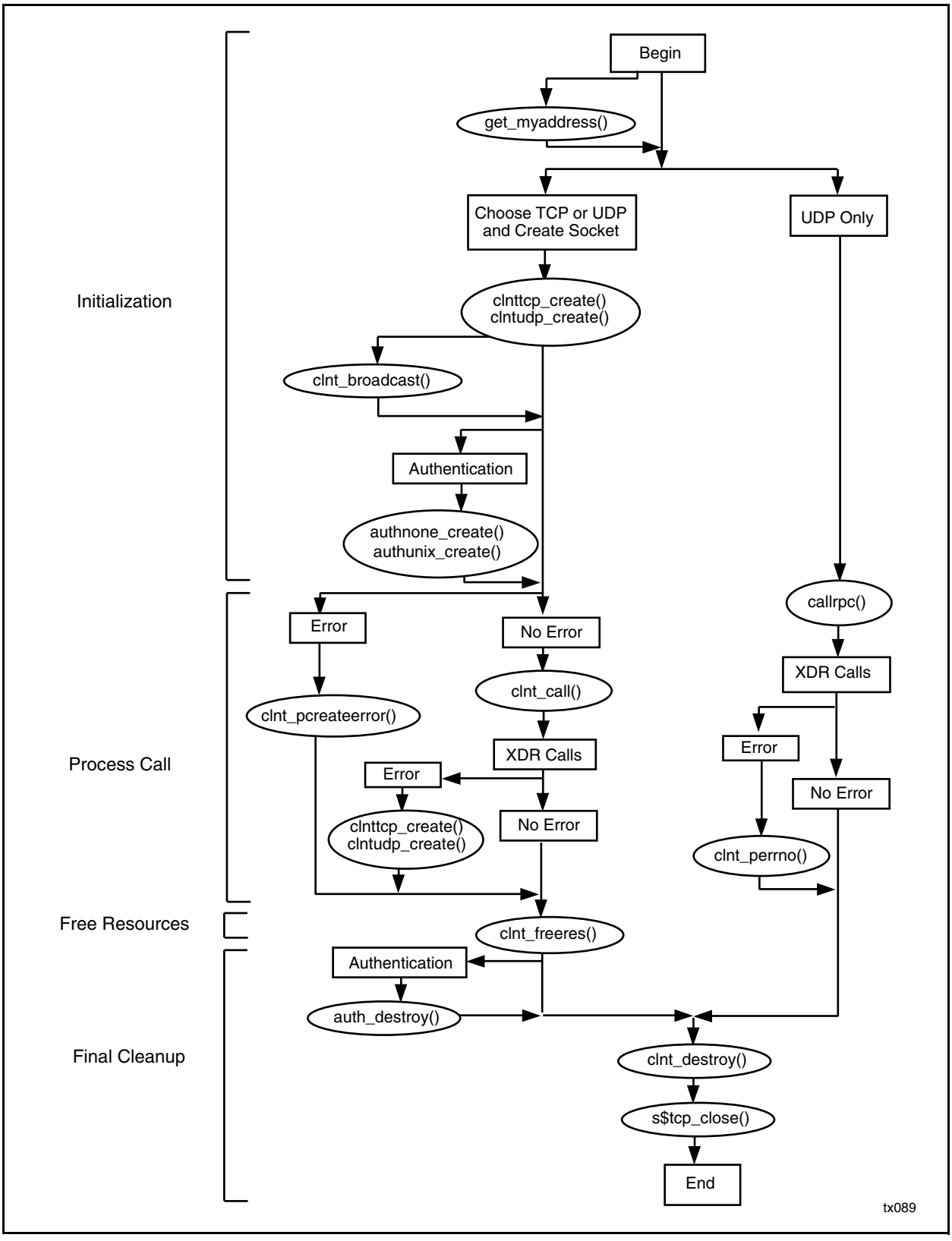

Figure 3-3. Flow Chart of the Function Calls Used on the RPC Client Side

The `callrpc()` function call is the simplest function call used to make remote procedure calls. However, you may need to use `clnt_call()` and its supporting RPC function calls instead of `callrpc()` so you can implement other tasks as discussed earlier in this chapter in the subsection "Choosing the RPC Function Calls."

In general, two arguments are passed with either the `callrpc()` or `clnt_call()` function calls. One is a pointer to the input data. The other is an XDR function call used to translate the input data to XDR so that it can be passed between various machine types. Also, two arguments are passed that handle the results. One is a pointer to the results, and the other is an XDR function call to translate the results from XDR.

The XDR function calls that are passed to translate the data are called *type field arguments*. For more information on type field arguments, see "XDR Function Calls" on page 3-15.

Using `callrpc()`

The `callrpc()` function call always uses UDP as the transport protocol and `RPC_ANYSOCK` as the socket. If `callrpc()` does not receive an answer after trying several times to deliver a message, it returns a return value.

The `callrpc()` function call has eight arguments:

- the name of the remote machine, `host`
- the program number, `prognum`, of the server to be called
- the version number, `versnum`, of the procedure to be called
- the procedure number, `procnum`, of the procedure to be called
- the XDR function call, `inproc`, which serializes the data for the input argument to XDR
- the pointer to the input data, `in`
- the XDR function call, `outproc`, which deserializes the data for the output argument from XDR
- the pointer to the output data, `out`.

If it completes successfully, `callrpc()` returns 0; if not, it will return a nonzero value. The meaning of the return values are found in the `clnt.h` include file, which resides in the directory `>system>rpc_include_library`.

Using Function Calls Instead of `callrpc()`

There are several function calls to use if you do not use `callrpc()`. The function call `clnt_call()` actually makes the RPC call and takes a `CLIENT` pointer rather than a host name.

There are seven arguments for `clnt_call()`:

- the `CLIENT` pointer, `client`, created by `clntudp_create()` or `clnttcp_create()`.
- the procedure number, `procnum`, of the procedure to be called.
- the XDR function call, `inproc`, which serializes the data for the input argument to XDR.
- the pointer to the input data, `in`.
- the XDR function call, `outproc`, which deserializes the data for the output argument from XDR.
- the pointer to the output data, `out`.
- the amount of time to wait for a reply. The argument, `tout` (time-out), is specified in seconds. The number of tries is the `clnt_call()` time-out divided by the `clntudp_create()` time-out.

To set up the client side of an RPC program, follow these steps.

1. Get the address of the host of the remote server.
2. Set up the time-out for use with the `clntudp_create()` function call.
3. Call `clntudp_create()` with the appropriate host address, the length of the server address, the program and version numbers, the value for time-out from Step 2, and the pointer to the socket. The `CLIENT` handle is created with the transport protocol specified.
4. Make the call to the remote procedure using `clnt_call()`.
5. Close the client handle using `clnt_destroy()`.
6. Close the socket using `s$tcp_close()` for TCP/IP Version 2 or `net_close()` for OS TCP/IP. For information on the `s$tcp_close()` function call, see the *VOS Communications Software: TCP/IP Programmer's Guide* (R129); for information on the `net_close()` function call, see the *VOS Communications Software: OS TCP/IP Programmer's Manual* (R224).

The `clnt_destroy()` call deallocates any space associated with the `CLIENT` handle, but does not close the socket associated with the `CLIENT` handle. This is because a socket can be reused for multiple client handles. Thus, if there are multiple client handles using the same socket, it is possible to close one handle without destroying the socket that other handles are using.

When the `clntudp_create()` call is made with an unbound socket, the system queries the Portmap server on the machine to which the call is being made and gets the appropriate port number. If the Portmap server is not running or has no port corresponding to the RPC call, the RPC call fails. Users can make RPC calls to the Portmap server themselves using the `pmap` function calls. For more information on the `pmap` function calls, see Chapter 4. For more information on selecting sockets, see the *VOS Communications Software: TCP/IP Programmer's Guide* (R129) for TCP/IP Version 2.

Using TCP in the Client Program

To use TCP as the transport protocol, use `clnttcp_create()` instead of `clntudp_create()` when creating the client transport handle. When the `clnttcp_create()` call is made, a TCP stream connection is established. All RPC calls using this `CLIENT` handle should reuse this connection. The general format of the `clnttcp_create()` function call is shown below.

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

Note that no time-out argument is used; instead, the receive and send buffer sizes must be specified.

All other client-side function calls are used as they are in the UDP protocol.

The General Structure of the RPC Server Side

This section describes the function calls used on the RPC server side of a program. Figure 3-4 presents a flowchart of the calls typically used on the RPC server side.

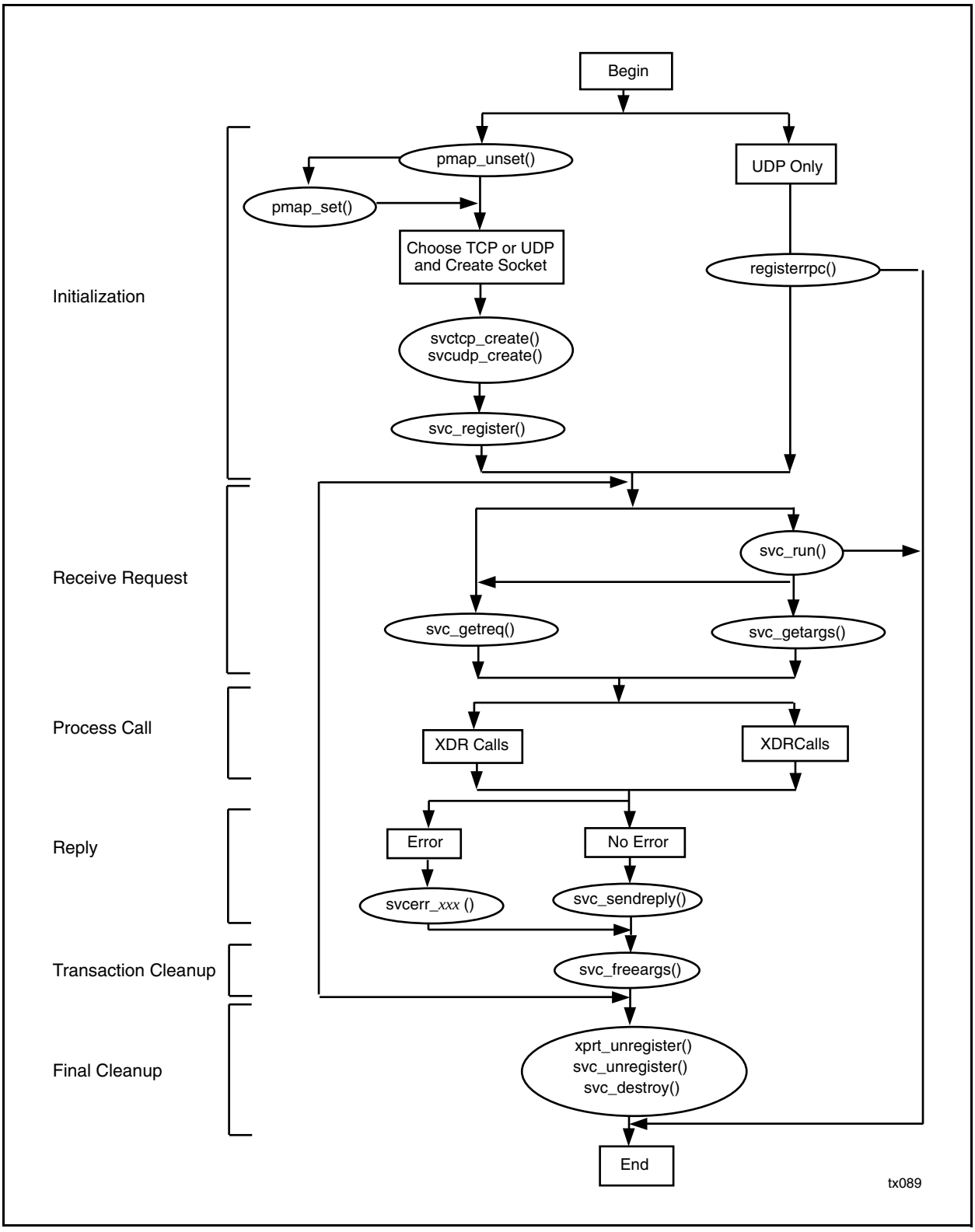

Figure 3-4. Flow Chart of the Function Calls Used on the RPC Server Side

The `registerrpc()` function call is the simplest function call used to register procedures with the Portmap server. However, you may need to use RPC function calls other than `registerrpc()` so that you can implement other tasks, as discussed earlier in this chapter in the section "Choosing the RPC Function Calls."

Using `registerrpc()`

The `registerrpc()` function call registers a C procedure within the server so that it corresponds with an RPC procedure number within the Portmap server. The `registerrpc()` function call always uses the transport protocol UDP. Also, `registerrpc()` handles registration at the procedure level, which means that each procedure must be registered individually by using multiple calls.

In general, two arguments are passed with `registerrpc()`. One is a type field argument used to translate the input data from XDR. The other is a type field argument to translate the results to XDR so that it can be passed between various machine types. For more information on type field arguments, see "XDR Function Calls" on page 3-15.

The `registerrpc()` function call has six arguments:

- the program number, `prognum`, of the server to be registered
- the version number, `versnum`, of the procedure to be registered
- the procedure number, `procnum`, of the procedure to be registered
- the procedure name, `procname`, of the procedure to be registered
- the XDR function call, `inproc`, which is used on the server side to deserialize the input data from XDR
- the XDR function call, `outproc`, which is used on the server side to serialize the output data to XDR.

If it completes successfully, `registerrpc()` returns 0; if not, it returns a nonzero value.

Using Function Calls Instead of `registerrpc()`

There are several function calls to use if you do not use `registerrpc()`. The function call that actually registers an RPC program is `svc_register()`. The `svc_register()` function call registers a program number rather than a procedure number.

There are five arguments for `svc_register()`:

- the `SVCXPRT` pointer, `xprt`, created by `svcudp_create()` or `svctcp_create()`. This is the RPC service transport handle.
- the program number, `prognum`, of the server to be registered.
- the version number, `versnum`, of the procedure to be registered.

- the dispatch procedure, `dispatch`, to be associated with program number `prognum` and version number `versnum`.
- the communication protocol, `protocol`, to be used. The value of `protocol` is usually `IPPROTO_UDP` or `IPPROTO_TCP`.

The following steps describe how to set up the server side of an RPC program when using `svcudp_create()` or `svctcp_create()`.

1. Set up the transport handle by calling `svcudp_create()` with the appropriate socket.
2. Clear the program number entry from the Portmap server's tables using `pmap_unset()`. This is done to erase any trace of the previous server program before restarting it, in case it was stopped earlier.
3. Register the program number with the Portmap server using `svc_register()`. This associates the program number with your dispatch procedure.
4. Call `svc_run()` to wait for a service request. When a request is received, the dispatch routine specified in `svc_register()` will be called. Your dispatch routine then calls the appropriate procedure based on the procedure number and version number.
5. If there is input data to be handled, call `svc_getargs()` to extract the input arguments from the call message for the service procedure's use. The function call `svc_getargs()` takes as arguments an `SVCXPRT` handle, the XDR function call to convert the input, and a pointer to where the input is to be placed.
6. Insert the code to perform the desired service.
7. Send a reply message to the client, using `svc_sendreply()`. The first argument is the `SVCXPRT` handle. The second argument is the XDR function call to serialize the results. The third argument is a pointer to the results to be returned.

If the argument to `svcudp_create()` is `RPC_ANYSOCK`, RPC creates a socket on which to send out RPC calls. Otherwise, `svcudp_create()` expects its argument to be a valid socket number. If the socket is bound to a port, the port numbers of `svcudp_create()` and `clntudp_create()` must match. When you specify `RPC_ANYSOCK` for a socket or give an unbound socket, the port number is determined when a server starts up. The server calls the local Portmap server. The Portmap server then chooses a port number for the RPC procedure if the socket specified to `svcudp_create()` is not already bound. For more information on selecting sockets, see the *VOS Communications Software: TCP/IP Programmer's Guide* (R129) for TCP/IP Version 2.

There are several differences between using `registerrpc()` and using `svc_register()`. Unlike `registerrpc()`, there are no XDR function calls involved in the registration process of `svc_register()`. Also, `dispatch_routine()` must call the appropriate C procedure calls based on the procedure number. Note that two things are handled by `dispatch_routine()` that `registerrpc()` handles automatically. First, procedure `NULLPROC` returns with no arguments, since, by convention, procedure 0 of any C routine takes no arguments and returns no results. This can be used as a simple test for detecting if a

remote program is running. Second, there is a check for invalid procedure numbers. If one is detected, `svcerr_noproc()` is called to handle the error.

Using TCP in the Server Program

To use TCP as the transport protocol, use `svctcp_create()` instead of `svcudp_create()` when creating the server transport handle. When the `svctcp_create()` call is made, a TCP stream connection is established. All RPC calls using this `SVCXPRT` handle should reuse this connection. The general format of the `svctcp_create()` function call is shown below.

```
svctcp_create(sock, send_buf_size, recv_buf_size)
```

Notice that you must specify the socket `sock`, as in `svcudp_create()`, and you must also specify the send and receive buffer sizes.

All other server-side function calls are used as they are in the UDP protocol.

Assigning Program Numbers

Program numbers are assigned in groups of `0x20000000` (536870912), as shown in Table 3-1.

Table 3-1. Assigning RPC Program Numbers

Group	Program Numbers	Description
1	`0 - 1fffffff`	Defined by manufacturers
2	`20000000 - 3fffffff`	Defined by user
3	`40000000 - 5fffffff`	Transient
4	`60000000 - 7fffffff`	Reserved
5	`80000000 - 9fffffff`	Reserved
6	`a0000000 - bfffffff`	Reserved
7	`c0000000 - dfffffff`	Reserved
8	`e0000000 - ffffffff`	Reserved

The first group of numbers is assigned by computer manufacturers. If you develop a commercial package, the appropriate program numbers should be assigned by the Stratus Customer Assistance Center (CAC). The second group of numbers is reserved for specific user programs. This group is intended primarily for programs that are site specific. The third group, transient, is reserved for programs that generate program numbers dynamically. For more information about programs that use the third group, see “Using Callback Procedures” on page 3-26. The final groups are reserved for future use and should not be used.

XDR Function Calls

Each XDR function call serializes and deserializes data and can be used to allocate memory. The process of converting from a particular machine representation to XDR is called *serializing*, and the reverse process is called *deserializing*. When an XDR function call is called from `callrpc()` or `clnt_call()`, the serializing part is used. When called from `svc_getargs()`, the deserializer and possibly the memory allocator is used. When called from `svc_freeargs()`, the memory deallocator is used. Using XDR function calls to allocate and deallocate memory is discussed in "Allocating Memory" later in this chapter.

RPC can pass various data structures, regardless of different machines' byte orders or structure layout conventions, by converting them to XDR before sending them to a client or server program. Before the data structures are used within the client or server program, they are deserialized from XDR to the machine-dependent representation. An XDR function call returns nonzero, true in the sense of C, if it completes successfully, and 0 otherwise.

Some XDR function calls are already defined for common data structures. These function calls are referred to as *predefined* function calls. A type field argument that is used with many function calls can be a predefined XDR function call or a user-supplied function call.

Predefined XDR Function Calls

Some predefined function calls can be used directly as type field arguments, while others cannot. The function calls that can be used directly as type field arguments are listed below.

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

The following is an example of the direct use of predefined function calls as type field arguments.

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM,
  xdr_u_short, &short_data, xdr_bool, &bool_data);
```

The following are also predefined XDR function calls, but they must be used within a function call that you have defined for your program, since they cannot be used directly. Defining your own XDR function calls is discussed in the next subsection, "User-Defined XDR Function Calls."

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

User-Defined XDR Function Calls

You may define XDR function calls that allow you to describe structures not found in the predefined function calls. The predefined function calls may be used within your user-defined function calls.

XDR always converts data to 4-byte multiples when deserializing. Thus, if there are characters in a structure instead of integers, each character will occupy 32 bits. Characters are handled by the XDR function call, `xdr_bytes()`, which is like `xdr_array()`, except that

it packs characters. The `xdr_bytes()` function call has four arguments, similar to the first four arguments of `xdr_array()`. For null-terminated strings, there is also the `xdr_string()` function call, which is the same as `xdr_bytes()` without the length argument. On serializing, it gets the string length from `strlen()`, and on deserializing it creates a null-terminated string.

To set up a user-defined function call, follow these steps.

1. Define the structure.
2. Write the function call to handle the new structure.
3. When using an RPC function call that requires you to pass the new structure, pass the user-defined function call as the type field argument and a pointer to the structure as the input or output argument.

Summary of Function Calls

The RPC and XDR function calls can be grouped by task. These tasks include authenticating RPC calls, implementing client-side routines, implementing server-side routines, interfacing with the Portmap server, error handling, and passing arbitrary data structures. See Chapter 4 for detailed information about these function calls.

Client Function Calls

The following function calls are used in the implementation of RPC client processes. Some of them manipulate the client handle, `CLIENT`. For more information on using these function calls, see "The General Structure of the RPC Client Side" on page 3-6.

`callrpc()`	Calls the remote procedure associated with `prognum`, `versnum`, and `procnum` on the remote machine
`clnt_call()`	Calls the remote procedure `procnum` associated with the client handle `clnt`
`clntudp_create()`	Creates an RPC client, which uses the UDP transport protocol, for the remote program `prognum` and version `versnum`
`clnttcp_create()`	Creates an RPC client, which uses the TCP transport protocol, for the remote program `prognum` and version `versnum` specified
`clnt_broadcast()`	Makes a broadcast remote procedure call to all locally connected broadcast networks
`clnt_destroy()`	Destroys the client's RPC handle
`clnt_freeres()`	Frees any data allocated by the `clnt_call()` function call when it decodes the arguments (results) of a reply message

Client Error Function Calls

The following function calls enable you to obtain information on error conditions following client operations. For information on using these function calls, see "The General Structure of the RPC Client Side" on page 3-6.

`clnt_geterr()`	Copies the error structure out of the client handle to the structure at address `errp`
`clnt_pcreateerror()`	Prints a message to default output indicating why an RPC handle creation function call failed
`clnt_perrno()`	Prints a message to default output corresponding to the condition indicated by `stat`, which is the same as `clnt_stat`
`clnt_perror()`	Prints a message to default output that indicates why an RPC call failed

Server Function Calls

The following function calls are used in the implementation of RPC server processes and manipulate the service transport handle `SVCXPRT`. For information on using these function calls, see "The General Structure of the RPC Server Side" on page 3-10.

(Page 1 of 2)

`registerrpc()`	Registers the procedure `procname` with the RPC service package
`svc_register()`	Associates the `prognum` and `versnum` arguments with the service dispatch procedure `dispatch()`
`svcudp_create()`	Creates a UDP-based RPC service transport, to which it returns a pointer
`svctcp_create()`	Creates a TCP-based RPC service transport, to which it returns a pointer
`svc_getargs()`	Decodes the arguments of a call message associated with the RPC service transport handle
`svc_freeargs()`	Frees any data allocated by the `svc_getargs()` function call when it was used to decode the arguments contained in the call message
`svc_getcaller()`	Gets the network address of the caller of a procedure associated with the RPC service transport handle
`svc_getreq()`	Services all the file descriptors represented in an `rdfds` bit mask, calling the proper dispatch routines
`svc_run()`	Waits for RPC requests to arrive and calls the appropriate service procedure, using `svc_getreq()`, when one arrives
`svc_sendreply()`	Sends the results of a remote procedure call when called by an RPC service's dispatch routine

(Page 2 of 2)

`svc_unregister()`	Removes all mapping of the double arguments, `prognum`, `versnum`, to dispatch routines, and removes all mapping of the triple arguments, `prognum`, `versnum`, `*`, to port number
`svc_destroy()`	Destroys the RPC service transport handle

Server Error Function Calls

The following function calls enable you to obtain information on error conditions following server operations. For information on using these function calls, see "The General Structure of the RPC Server Side" on page 3-10.

`svcerr_auth()`	Called by a service dispatch routine that cannot perform a remote procedure call due to an authentication error
`svcerr_decode()`	Called by a service dispatch routine that is unable to successfully decode its arguments; see also `svc_getargs()` in the preceding subsection "Server Function Calls"
`svcerr_noproc()`	Called by a service dispatch routine that does not support the procedure number the caller requested
`svcerr_systemerr()`	Called by a service dispatch routine when it detects a system error not covered by any existing protocol
`svcerr_weakauth()`	Called by a service dispatch routine that cannot perform a remote procedure call due to insufficient, but correct, authentication arguments

Portmap Interface Function Calls

The following function calls enable you to call the Portmap server directly. For information on using the `pmap_set()` and `pmap_unset()` function calls, see "Using Function Calls Instead of registerrpc()" on page 3-12.

(Page 1 of 2)

`pmap_getmaps()`	User interface to the Portmap server that returns a list of the current RPC program-to-port mappings on the host located at Internet Protocol (IP) address `*addr`
`pmap_getport()`	User interface to the Portmap server that returns the port number on which a server is waiting, given the server's program number `prognum`, version number `versnum`, and transport protocol associated with `protocol`
`pmap_rmtcall()`	User interface to the Portmap server that instructs the Portmap server on the host at IP address `*addr` to make an RPC call to a procedure on that hostUser interface to the Portmap server that instructs the Portmap server on the host at IP address `*addr` to make an RPC call to a procedure on that host

(Page 2 of 2)

`pmap_set()`	User interface to the Portmap server that establishes a mapping between the triple arguments `prognum, versnum, protocol`, and `port` on the machine's Portmap server
`pmap_unset()`	User interface to the Portmap server that destroys mapping between the triple arguments `prognum, versnum, *`, and `port` on the machine's Portmap server

Authentication Function Calls

The following function calls manipulate the authentication handle, `auth`, and the required credentials. For information on using these function calls, see ''Incorporating Authentication'' on page 3-21.

`authnone_create()`	Creates and returns an RPC authentication handle that passes authentication information of the type `AUTH_NULL`
`authunix_create()`	Creates and returns an RPC authentication handle that contains UNIX authentication information of the type `AUTH_NULL`
`auth_destroy()`	Destroys the authentication information associated with the authentication handle, `auth`

Predefined XDR Function Calls

The following function calls enable you to pass data structures in their standard XDR external representations between client and server processes. For information on using these function calls, see ''XDR Function Calls'' on page 3-15.

(Page 1 of 2)

`xdr_int()`	A filter primitive that translates between C integers and their external representations
`xdr_long()`	A filter primitive that translates between C long integers and their external representations
`xdr_short()`	A filter primitive that translates between C short integers and their external representations
`xdr_u_int()`	A filter primitive that translates between C unsigned integers and their external representations
`xdr_u_long()`	A filter primitive that translates between C unsigned long integers and their external representations
`xdr_u_short()`	A filter primitive that translates between C unsigned short integers and their external representations
`xdr_float()`	A filter primitive that translates between C floats and their external representations
`xdr_double()`	A filter primitive that translates between C double precision numbers and their external representations

(Page 2 of 2)

`xdr_enum()`	A filter primitive that translates between C `enums` and their external representation
`xdr_bool()`	A filter primitive that translates between Boolean values and their external representations
`xdr_string()`	A filter primitive that translates between C strings and their corresponding external representations
`xdr_wrapstring()`	Calls `xdr_string(xdrs, sp, MAXUNSIGNED)`, where `MAXUNSIGNED` is the maximum value of an unsigned integer
`xdr_bytes()`	A filter primitive that translates between counted byte strings and their external representations
`xdr_array()`	A filter primitive that translates between arrays and their corresponding external representations
`xdr_opaque()`	A filter primitive that translates between fixed-size opaque data and its external representation
`xdr_union()`	A filter primitive that translates between a discriminated C union and its corresponding external representation
`xdr_reference()`	Provides pointer chasing within structures
`xdr_void()`	Used to convert a void result

Allocating Memory and Incorporating Authentication

This section discusses the RPC procedures and function calls used to allocate memory and incorporate authentication.

Allocating Memory

XDR function calls not only serialize and deserialize data, they can also allocate memory. If the pointer to a structure is `NULL`, then the XDR function call allocates space for the structure and returns a pointer to it, putting the size of the structure in the third argument.

To designate an XDR function call to allocate memory, follow these steps.

1. Pass a null pointer to the `svc_getargs()` function call. The `svc_getargs()` function then passes the pointer to the XDR function call.

2. Call the `svc_freeargs()` function call to deallocate the memory when you are finished with the allocated structure.

Incorporating Authentication

The RPC protocol provides authentication function calls that maintain the fields necessary for a client to identify itself to a server. These fields are referred to as *credentials*. The authentication function calls are listed below.

- `authnone_create()`
- `authunix_create()`
- `auth_destroy()`

The authentication function calls are used by a client program to create the requested credentials and then destroy those credentials when they are no longer needed. The credentials are passed within the call message to the server program, which determines whether the call message will be accepted or rejected. Thus, individual servers must implement their own access control policies and reflect these policies as return values in their reply messages.

Two types of authentication are supported by RPC. The `AUTH_NULL` authentication type is the default type used if no authentication is specified or wanted. The `AUTH_UNIX` authentication type is used when the caller of a remote procedure wants to identify itself. The `AUTH_UNIX` authentication uses the same credentials, such as `UID` and `GID`, that a user on a UNIX system uses for identification.

The Client Side

When using `callrpc()`, a new client handle is created automatically and its authentication type defaults to `AUTH_NULL`. However, if you are using `clnt_call()` and its supporting function calls, you can choose the type of authentication you want. When a client creates a new RPC client handle, as shown in the following `clntudp_create()` example, the transport protocol sets the associated authentication handle to `AUTH_NULL`.

```
clnt = clntudp_create(address, prognum, versnum, wait, &sock)
```

To implement UNIX-style authentication, the `authunix_create()` function call may be used after the client handle is created. This function call will cause any subsequent RPC call associated with `clnt` to carry the following authentication credentials structure.

```
struct authunix_parms
     {
     u_long      aup_time;          /* credentials creation time */
     char        *aup_machname;     /* host name where client is */
     int        aup_uid;          /* client's \unix\ effective uid */
     int         aup_gid;           /* client's current group id */
     u_int       aup_len;           /* element length of aup_gids */
     int         *aup_gids;         /* array of groups user is in */
     };
```

After creating the client handle, set `clnt->cl_auth` with the `authunix_create()` function call. You must set up the credentials information structure `authunix_parms` in the client program. Using `authunix_create()` to set `clnt->cl_auth` is shown below.

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

The server program sends return values in its reply message. The client acts upon any return values.

Unless the AUTH_NULL authentication type is being used, you must destroy the authentication credentials structure when you are finished with it. This should always be done to conserve memory. The following function call destroys the authentication information.

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

The Server Side

It is difficult for authentication to be implemented on the server side because the server does not know what style of authentication the call message will be in. To determine the type of credentials the server will look for, you should examine the field rq_cred.oa_flavor, which is shown below.

```
/*
 * Authentication info. Mostly opaque to the programmer.
 */
struct opaque_auth
     {
     enum_t     oa_flavor;     /* style of credentials */
    caddr_t   oa_base;     /* pointer to |authunix_parms| structure
*/
     u_int      oa_length;     /* not to exceed MAX_AUTH_BYTES */
     };
```

The rq_cred.oa_flavor field can be found by means of a pointer in the svc_req structure.

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

The RPC protocol ensures that the request's rq_cred field is well formed. Thus, the service implementor may inspect the request's rq_cred.oa_flavor to determine which style of authentication the caller used. The RPC protocol also ensures that the request's rq_clntcred field is either NULL or points to a well-formed AUTH_UNIX credentials structure. The rq_clntcred field could be cast to a pointer to an authunix_parms structure.

The authentication arguments associated with NULLPROC (procedure number 0) are not checked. If the authentication argument's type is not suitable for the server in question, svcerr_weakauth() should be called. Also, the server protocol itself should return status

for access denied. If the server does not have the return value set up in its reply, the server error handler, `svcerr_systemerr()`, should be called instead.

Bypassing `svc_run()`

If your application program must wait for a file descriptor in order to update a data structure, while it is attempting to execute RPC requests, the standard `svc_run()` function will not work. To avoid this situation, you can call `svc_getreq()` directly by using the following code segment rather than calling `svc_run()`. This code segment provides a framework for specific user requirements and is not intended to be fully functional as written.

```
#include <tcp_socket.h>

svc_run()

{
     int       readfds;

     sel       sel_arr;
     int       nfound,cnt;
     u32       *waitp;
     u32       *vos_event_ids;
     u32       *vos_event_cnts;
     int       vos_number_events;
     u16       *vos_event_number;

    /* The vos_event_ids, vos_event_cnts and vos_number_events should
      * be set up to reflect all VOS events you want to wait on.
      */
               .
               .
               .
               .

     for(;;)
          {
          readfds = svc_fds;

          cnt = 32;  /* number of select structures in sel_arr */

          nfound = s$tcp_nselect_with_events( sel_arr, cnt, waitp,
                                        vos_event_ids, vos_event_cnts,
                                                vos_number_events,
                                                vos_event_number);
```

(Continued on next page)

```
                switch( nfound)
                     {

                case -2:

                     /* The return value of -2 indicates that a VOS event
                    * notification has occurred. vos_event_number contains
                    * the index of the notified event, and the appropriate
                      * vos_event_counts will be updated with the new
                      * event count.
                      *
                    * Perform appropriate actions to process the VOS event
                      * at this point.......
                      */

                     break;

              case -1:      /* Got an error on s$tcp_nselect_with_events */

                     if (errno == EINTR)
                          continue;
                     perror("rstat: select");
                     return;

                case 0:   /* Timeout occurred */

                     break;

             default:  /* otherwise the socket descriptor that was notified
*/

                     /* perform processing required to get the request */

                     svc_getreq(readfds);

                     }
                }
}
```

Batching and Broadcasting

This section discusses batching and broadcasting procedures for RPC programs.

Batching

The RPC architecture is designed so that a client sends a call message and waits for a server to reply that the call succeeded. The client does not compute while a server is processing a call. If the client does not want or need an acknowledgment for every message sent, it could continue computing while waiting for a response. RPC messages can be placed in a pipeline of calls to a desired server; this process is called *batching*.

To enable a server to handle batched messages, the following conditions must be met.

- Each RPC call in the pipeline must not require a response from the server, and the server must not send a reply message.
- The pipeline of calls must be transported using TCP.

For a client to take advantage of batching, the client must perform RPC calls using TCP and the calls must have the following attributes.

- The result's XDR function call must be 0 (`NULL`).
- The RPC call's time-out must be 0.

Since the server does not respond to every call, the client can generate new calls while the server is executing previous calls. Furthermore, the TCP implementation can place many call messages in a buffer and send them to the server in one `send()` system call. This overlapped execution greatly decreases the interprocess-communication overhead of the client and server processes and the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually execute a call waiting for a reply in order to flush the pipeline. Also, since the server sends no message, the client cannot be notified of any of the failures that may occur. Therefore, clients must implement their own error handling.

Broadcasting

A list of the main differences between broadcast RPC and normal RPC calls follows.

- Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answers from each responding machine).
- Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UDP/IP.
- Broadcast RPC filters out all unsuccessful responses. Thus, if there is a version mismatch between the broadcaster and a remote server, the user of broadcast RPC never knows.
- All broadcast messages are sent to the `portmap` port. Thus, only servers that register themselves with their Portmap server are accessible through the broadcast RPC mechanism.

An example of a broadcast RPC routine follows.

```
#include pmap_clnt.h

enum clnt_stat     clnt_stat;

clnt_stat = clnt_broadcast(prog, vers, proc, xargs, argsp, xresults,
  resultsp, eachresult)
u_long          prog;              /* program number */
u_long          vers;              /* version number */
u_long          proc;              /* procedure number */
xdrproc_t       xargs;             /* xdr routine for args */
caddr_t         argsp;             /* pointer to args */
xdrproc_t       xresults;          /* xdr routine for results */
caddr_t         resultsp;          /* pointer to results */
bool_t         (*eachresult)();   /* called with each result received
*/
```

The procedure `eachresult()` is called each time a valid result is obtained. It returns a Boolean value that indicates whether the client expects more responses.

```
bool_t done;

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

If `done` is `TRUE`, broadcasting stops and `clnt_broadcast()` returns successfully. Otherwise, the function call waits for another response. The request is rebroadcast after a few seconds. If no responses come back, the function call returns with `RPC_TIMEDOUT`. To interpret `clnt_stat` errors, supply the `clnt_perrno()` function call with the error code.

Callback Procedures and Support for Multiple Versions

This section describes how to use callback procedures and support multiple versions of a program.

Using Callback Procedures

Occasionally, it is useful to have a server become a client and call back the process which is its client. To do an RPC callback, a program number is needed to make the RPC call on. Since this will be a dynamically generated program number, it should be in the transient range `0x40000000` to `0x5fffffff`. For more information about program numbers, see "Assigning Program Numbers" on page 3-14. A routine that returns a valid program number in the transient range and registers it with the Portmap server is required. The steps involved in writing the routine follow.

1. Obtain a socket.
2. Bind the socket.
3. Obtain the assigned port.

4. Call `pmap_set()` using the above information until a transient program number is assigned. The call to `pmap_set()` is a test and set operation; that is, it indivisibly tests whether a program number has already been registered, and if it has not, the call then reserves it. On return, the `sockp` argument will contain a socket that can be used as the argument to an `svcudp_create()` or `svctcp_create()` call.

After setting up the routine to obtain the transient number, a server can call back its client process using the following steps as guidelines.

1. The client program gets a transient program number from `pmap_set`, as described above.

2. The client program calls `svcudp_create()` or `svctcp_create()` using the transient program number.

3. The client program makes an RPC call to the server, passing it the transient program number. The client then waits to receive a callback from the server at that program number. The server registers the program so that it can receive the RPC call informing it of the callback program number.

4. The server then sends a callback RPC call using the transient program number it received earlier.

Supporting Multiple Versions of a Program

To support multiple versions of a program, you must call `svc_register()` for each version. By doing this, the same C procedure will contain the versions under separate `case` statements.

Chapter 4:
RPC and XDR Function Calls

All RPC and XDR function calls are constructed according to the OpenVOS C programming language conventions. If you are writing a program in a OpenVOS programming language other than C, see the corresponding OpenVOS language manual for information on how to call a C function in that language.

This chapter documents the format and operation of the following standard RPC and XDR function calls. In describing the function calls and their arguments, references are made to include files located in the directory `>system>rpc_include_library`. All function calls are listed in alphabetical order.

- `auth_destroy()`
- `authnone_create()`
- `authunix_create()`
- `callrpc()`
- `clnt_broadcast()`
- `clnt_call()`
- `clnt_destroy()`
- `clnt_freeres()`
- `clnt_geterr()`
- `clnt_pcreateerror()`
- `clnt_perrno()`
- `clnt_perror()`
- `clnttcp_create()`
- `clntudp_create()`
- `get_myaddress()`
- `pmap_getmaps()`
- `pmap_getport()`
- `pmap_rmtcall()`
- `pmap_set()`
- `pmap_unset()`
- `registerrpc()`
- `rpc_errmsg()`
- `svc_destroy()`
- `svc_freeargs()`
- `svc_getargs()`
- `svc_getcaller()`
- `svc_getreq()`
- `svc_register()`
- `svc_run()`

- `svc_runable()`
- `svc_sendreply()`
- `svc_unregister()`
- `svcerr_auth()`
- `svcerr_decode()`
- `svcerr_noproc()`
- `svcerr_systemerr()`
- `svcerr_weakauth()`
- `svctcp_create()`
- `svcudp_create()`
- `xdr_array()`
- `xdr_bool()`
- `xdr_bytes()`
- `xdr_double()`
- `xdr_enum()`
- `xdr_float()`
- `xdr_int()`
- `xdr_long()`
- `xdr_opaque()`
- `xdr_reference()`
- `xdr_short()`
- `xdr_string()`
- `xdr_u_int()`
- `xdr_u_long()`
- `xdr_u_short()`
- `xdr_union()`
- `xdr_void()`
- `xdr_wrapstring()`

Function Call Return Values and Error Codes

Function call return values generally indicate the success or failure of the requested operation. In general, for a return a value of the type `bool_t`, the value 1 indicates success and the value 0 indicates failure.

For some calls, however, 0 indicates success, and a positive value indicates failure. For example, many function calls return enumeration constants of the type `enum clnt_stat`. In this case, an enumeration constant with a positive value indicates an error. To interpret return values of the type `enum clnt_stat`, call the function `clnt_perrno()`, which prints a message indicating the meaning of the enumeration constant. (See the `clnt_perrno()` function call description later in this chapter.) Table 4-1 describes the return values of `clnt_stat`.

Table 4-1. Return Values of `clnt_stat` *(Page 1 of 2)*

Enumeration Constant	Value	Description
RPC_SUCCESS	0	The call was successful.
RPC_CANTENCODEARGS	1	The client or the server cannot encode arguments, possibly due to invalid parameters passed with an XDR function.
RPC_CANTDECODERES	2	The client or the server cannot decode results, possibly due to a coding error or invalid parameters passed with an XDR function call.
RPC_CANTSEND	3	The client cannot send the call. This error message may have been generated by a TCP send operation. The user-specified buffer size for sending data may be too small, the host name may be incorrect, or the Ethernet network may not be functioning.
RPC_CANTRECV	4	The client cannot receive the result. This error message may have been generated by a TCP receive operation. The user-specified buffer size for receiving data may be too small, the host name may be incorrect, or the Ethernet network may not be functioning.
RPC_TIMEDOUT	5	The client did not receive a response in the specified time-out period. The remote system may not be operating or the server may be too busy to receive messages.
RPC_VERSMISMATCH	6	The RPC versions on the client side and server side do not match. The solution is to install the correct version of RPC.
RPC_AUTHERROR	7	The authentication call has an error.
RPC_PROGUNAVAIL	8	The server was unable to locate the remote program number.
RPC_PROGVERSMISMATCH	9	The version numbers of the programs on the client side and server side do not match.
RPC_PROCUNAVAIL	10	The server does not recognize the procedure that the client requested.
RPC_CANTDECODEARGS	11	The client or the server cannot decode arguments, possibly due to a coding error or invalid parameters passed with an XDR function call.
RPC_SYSTEMERROR	12	Either the program could not allocate additional memory or it failed to select a socket.
RPC_UNKNOWNHOST	13	The client did not recognize the specified host address or host name.
RPC_PMAPFAILURE	14	The RPC call to the Portmap server failed, possibly because the Portmap server is not running on the server or because of problems on the TCP or Ethernet network.

Table 4-1. Return Values of `clnt_stat` *(Page 2 of 2)*

Enumeration Constant	Value	Description
RPC_PROGNOTREGISTERED	15	The remote program is not registered.
RPC_FAILED	16	This error is unspecified.

The values that each function call can return are listed in the appropriate function call description.

Authentication errors are reported in an enumeration constant of the type `enum auth_stat`. You can determine the value of the `auth_stat` constant by calling `svcerr_auth()`. The value of the `why` argument of the `svcerr_auth()` function call equals the value of `auth_stat`. Table 4-2 lists the status labels and status codes for the `auth_stat` constant.

Table 4-2. Authentication Status Labels and Status Codes for `auth_stat`

Status Label	Status Code	Description	Failure Type
AUTH_OK	0	Credentials accepted	
AUTH_BADCRED	1	Bad credentials (seal broken)	Remote
AUTH_REJECTEDCRED	2	Client should begin new session	Remote
AUTH_BADVERF	3	Bad verifier (seal broken)	Remote
AUTH_REJECTEDVERF	4	Verifier expired or was replayed	Remote
AUTH_TOOWEAK	5	Rejected due to security reasons	Remote
AUTH_INVALIDRESP	6	Bad response verifier	Local
AUTH_FAILED	7	Some unknown reason	Local

RPC Global Variables

Several RPC and XDR function calls use two global variables, `rpc_createerr` and `svc_fds`. This section describes each of these global variables in detail.

The `rpc_createerr` Global Variable

The `rpc_createerr` global variable, which is defined in the `clnt.h` include file, has the following format.

```
struct rpc_createerr     rpc_createerr;
```

When the function calls `clnttcp_create()` and `clntudp_create()` fail, they set the global variable `rpc_createerr` to a value that indicates the type of error causing the failure. To determine the value stored in `rpc_createerr`, call the function `clnt_pcreateerror()`. For more information, see the `clnt_pcreateerror()` function call description later in this chapter.

The `svc_fds` Global Variable

The `svc_fds` global variable, which is defined in the `svc.h` include file, has the following format.

```
int     svc_fds;
```

The value of the `svc_fds` global variable indicates the read file-descriptor bit mask of the RPC server side. This variable is set only when a server processes its own asynchronous events rather than calling `svc_run()`. The `svc_fds` variable is read-only; however, the value of the variable may change after calls to `svc_getreq()` or any creation functions. In the section "Allocating Memory and Incorporating Authentication" in Chapter 3, the subsection "Bypassing `svc_run()`" presents a code sample that illustrates one use of the `svc_fds` global variable.

auth_destroy()

Purpose

The `auth_destroy()` function call destroys the authentication information associated with `auth`.

Format

```
#include <auth.h>

void auth_destroy(auth)

AUTH *auth;
```

Arguments

- `auth` (input/output)
  An authentication handle that points to a structure defined with the type `AUTH`. This handle is created by `authnone_create()` and `authunix_create()`. The `AUTH` type is defined in the `auth.h` include file.

Explanation

The `auth_destroy()` function call destroys the authentication information associated with `auth`. Destruction usually involves deallocation of private data structures. The use of `auth` is undefined after a call to `auth_destroy()`.

authnone_create()

Purpose

The `authnone_create()` function call creates and returns an RPC authentication handle that passes authentication information of the type `AUTH_NULL`.

Format

```
#include <auth.h>

AUTH * authnone_create()
```

Return Values

If successful, `authnone_create()` returns a valid pointer to a structure defined with the type `AUTH`. If unsuccessful, `authnone_create()` returns `NULL`. The `AUTH` type is defined in the `auth.h` include file.

authunix_create()

Purpose

The `authunix_create()` function call creates and returns an RPC authentication handle that contains UNIX authentication information.

Format

```
#include <auth.h>

AUTH * authunix_create(host, uid, gid, len, aup_gids)

char *host;
int uid;
int gid;
int len;
int *aup_gids;
```

Arguments

- `host` (input)
  The Internet host name, as found in the host machine database file of the machine on which the information was created. For TCP/IP Version 2, the host machine database file is the `machine.db` file, which resides in the directory `>system>tcp`.

- `uid` (input)
  The user's UNIX user ID.

- `gid` (input)
  The user's current UNIX group ID.

- `len` (input)
  The number of groups to which a user belongs.

- `aup_gids` (input)
  A counted array of groups to which the user belongs. This consists of a variable one-dimensional array of integers.

Return Values

If successful, `authunix_create()` returns a valid pointer to a structure defined with the type `AUTH`. If unsuccessful, `authunix_create()` returns `NULL`. The `AUTH` type is defined in the `auth.h` include file.

callrpc()

Purpose

The `callrpc()` function call, which uses UDP, makes a call to the remote procedure associated with `prognum`, `versnum`, and `procnum` on the machine `host`.

Format

```
#include <clnt.h>

bool_t callrpc(host, prognum, versnum, procnum, inproc, in, outproc,
out)

char *host;
unsigned long prognum;
unsigned long versnum;
unsigned long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
```

Arguments

- `host` (input)
  The Internet host name, as found in the host machine database file of the machine on which the information was created. For TCP/IP Version 2, the host machine database file is the `machine.db` file, which resides in the directory `>system>tcp`.

- `prognum` (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range `20000000` to `3fffffff` hexadecimal.

- `versnum` (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program may support multiple versions. The value of `versnum` should be greater than 0.

- `procnum` (input)
  The procedure number within the requested service and version.

- `inproc` (input)
  A pointer to a procedure as defined by the `xdrproc_t` type definition. This procedure converts the `in` data to XDR format. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The function `xdrproc_t`, defined in the

`xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- `in` (input)
  A pointer to character data that contains the procedure's arguments.

- `outproc` (input)
  A pointer to a procedure as defined by the `xdrproc_t` type definition. This procedure converts the data from the result of the call that is in XDR format to host-native format, which is stored in `out`. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The function `xdrproc_t`, defined in the `xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- `out` (output)
  A pointer to character data containing the results of the call.

Explanation

When calling remote procedures, this function call uses UDP as the transport protocol. See the description of the `clntudp_create()` function call for restrictions.

When creating transport handles, this function call uses the field `RPC_ANYSOCK`.

Return Values

The `callrpc()` function call returns 0 if successful. If unsuccessful, the `callrpc()` function call returns a value of the type `enum clnt_stat` converted to an `int`. (The function call `clnt_perrno()` prints the enumerated definition of the value stored in `clnt_stat`.) The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file. See Table 4-1 for a list of the return values.

clnt_broadcast()

Purpose

The `clnt_broadcast()` function call makes a broadcast remote procedure call to all locally connected broadcast networks.

Format

```
#include <pmap_clnt.h>

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

Arguments

- `prognum` (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range `20000000` to `3fffffff` hexadecimal.

- `versnum` (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of `versnum` should be greater than 0.

- `procnum` (input)
  The procedure number within the requested service and version.

- `inproc` (input)
  A pointer to a procedure as defined by the `xdrproc_t` type definition. This procedure converts the `in` data to XDR format. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The function `xdrproc_t`, defined in the `xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- `in` (input)
  A pointer to character data that contains the procedure's arguments.

- `outproc` (input)
  A pointer to a procedure as defined by the `xdrproc_t` type definition. This procedure converts the data from the result of the call that is in XDR format to host-native format, which is stored in `out`. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The function `xdrproc_t`, defined in the `xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- `out` (output)
  A pointer to character data containing the results of the call.

- `eachresult` (input)
  A pointer to a procedure as defined by the type definition `resultproc_t`, which is defined in `clnt.h`. See "Explanation" below.

Explanation

The `clnt_broadcast()` function call is like `callrpc()` except that the call message is broadcast to all locally connected broadcast networks. Each time it receives a response, the `clnt_broadcast` function calls `eachresult()`, whose format is as follows:

```
eachresult(out, addr)

char *out;
struct sockaddr_in *addr;
```

- `out` (input)
  A pointer to character data containing the results of the call. This pointer is the same as the `out` argument passed to `clnt_broadcast()` except that the remote procedure's output is decoded there.

- `addr` (input)
  A pointer to the address of the machine that sent the results.

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

Return Values

Table 4-1 lists the return values returned by `clnt_broadcast()` using `clnt_stat`.

clnt_call()

Purpose

The clnt_call() function calls the remote procedure procnum associated with the client handle clnt.

Format

```
#include <clnt.h>

enum clnt_stat clnt_call(clnt, procnum, inproc, in, outproc, out,
tout)

CLIENT *clnt;
long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
```

Arguments

- clnt (input)
  A handle to a CLIENT type definition as defined in the clnt.h include file. This handle is created by clnttcp_create() or clntudp_create(). It is destroyed by clnt_destroy().

- procnum (input)
  The procedure number within the requested service and version.

- inproc (input)
  A pointer to a procedure as defined by the xdrproc_t type definition. This procedure converts the in data to XDR format. See, for example, xdr_int(), xdr_long(), xdr_char(), and other XDR function calls. The function xdrproc_t, defined in the xdr.h include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- in (input)
  A pointer to character data that contains the procedure's arguments.

- outproc (input)
  A pointer to a procedure as defined by the xdrproc_t type definition. This procedure converts the data from the result of the call that is in XDR format to host-native format, which is stored in out. See, for example, xdr_int(), xdr_long(), xdr_char(),

and other XDR function calls. The function `xdrproc_t`, defined in the `xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- `out` (output)
  A pointer to character data containing the results of the call.

- `tout` (input)
  A structure containing time-out information in seconds and microseconds as defined in `timeval.h`. The `tout` argument is the time allowed for results to come back.

Return Values

The `clnt_call()` function call returns the enumerated type `clnt_stat`, as described in Table 4-1.

clnt_destroy()

Purpose

The `clnt_destroy()` function call destroys the client's RPC handle.

Format

```
#include <clnt.h>

clnt_destroy(clnt)

CLIENT *clnt;
```

Arguments

- `clnt` (input)
  A handle to a `CLIENT` type definition as defined in `clnt.h`. This handle is created by `clnttcp_create()` or `clntudp_create()`. It is used by `clnt_call()` and `clnt_broadcast()`.

Explanation

Destruction usually involves deallocation of private data structures, including `clnt` itself. Use of `clnt` is undefined after a call to `clnt_destroy()`. It is the user's responsibility to close sockets associated with `clnt`.

clnt_freeres()

Purpose

The `clnt_freeres()` function call frees any data allocated by the RPC/XDR system during the decoding of the results of an RPC call.

Format

```
#include <clnt.h>

clnt_freeres(clnt, outproc, out)

CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

Arguments

- `clnt` (input)
  A handle to a `CLIENT` type definition as defined in `clnt.h`. This handle is created by `clnttcp_create()` or `clntudp_create()`. It is used by `clnt_call()` and `clnt_broadcast()`.

- `outproc` (input)
  A pointer to a procedure as defined by the `xdrproc_t` type definition, which is an XDR routine describing the results in simple primitives. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The function `xdrproc_t`, defined in the `xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- `out` (input)
  A pointer to the results of a decoded RPC call.

Return Values

The `clnt_freeres()` function call returns 1 if successful and 0 if unsuccessful.

clnt_geterr()

Purpose

The `clnt_geterr()` function call copies the error structure from the client handle to the structure at address `errp`.

Format

```
#include <clnt.h>

void clnt_geterr(clnt, errp)

CLIENT *clnt;
struct rpc_err *errp;
```

Arguments

- `clnt` (input)
  A handle to a `CLIENT` type definition, as defined in `clnt.h`. This handle is created by `clnttcp_create()` or `clntudp_create()`. It is used by `clnt_call()` and `clnt_broadcast()`.

- `errp` (output)
  A pointer to a structure, as defined in the `clnt.h` include file.

clnt_pcreateerror()

Purpose

The `clnt_pcreateerror()` function call prints a message to default output indicating why an RPC handle-creation routine returned `NULL`, which indicates failure.

Format

```
#include <clnt.h>

void clnt_pcreateerror(s)

char *s;
```

Arguments

- `s` (input/output)

  An error-message string created by the user that is implementation specific. This string should not be more than 32 characters long.

Explanation

The `clnt_pcreateerror()` function call prints out the message stored in the global variable `rpc_createerr`. The message that `clnt_pcreateerror()` returns is prefixed by the user-defined string `s` and a colon. The `clnt_pcreateerror()` function call is used after the `clnttcp_create()` or `clntudp_create()` function call and checks for the errors listed in Table 4-1 and Table 4-2.

clnt_perrno()

Purpose

The `clnt_perrno()` function call prints a message to default output corresponding to the value of `stat`, which equals the value `clnt_stat`.

Format

```
#include <clnt.h>

void clnt_perrno(stat)

enum clnt_stat stat;
```

Arguments

- `stat` (input)
  An enumerated type, as described in Table 4-1.

Explanation

The `clnt_perrno()` function call is used after the `callrpc()` function call.

clnt_perror()

Purpose

The `clnt_perror()` function call prints a message to default output that indicates why an RPC call failed.

Format

```
#include <clnt.h>

void clnt_perror(clnt, s)

CLIENT *clnt;
char *s;
```

Arguments

- `clnt` (input)
  A handle to a `CLIENT` type definition, as defined in the `clnt.h` include file. This handle is created by `clnttcp_create()` or `clntudp_create()`. It is used by `clnt_call()` and `clnt_broadcast()`.

- `s` (input/output)
  An error-message string created by the user that is implementation specific. This string should not be more than 32 characters long.

Explanation

The `clnt_perror()` function call is used after the `clnt_call()` function call.

clnttcp_create()

Purpose

The clnttcp_create() function call creates an RPC client for the remote program prognum and version versnum.

Format

```
#include <clnttcp_create>

CLIENT * clnttcp_create(addr, prognum, versnum, sockp, sendsz,
recvsz)

struct sockaddr_in *addr;
unsigned long prognum;
unsigned long versnum;
int *sockp;
unsigned int sendsz;
unsigned int recvsz;
```

Arguments

- addr (input/output)
  The address of the Internet-style socket address information. This structure is in the in.h include file. If the port number is 0, a binder on the remote machine is consulted for the correct port number.

- prognum (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range 20000000 to 3fffffff hexadecimal.

- versnum (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of versnum should be greater than 0.

- sockp (input/output)
  A pointer to a requested socket. If the socket value is less than 0, the socket is set to a newly created TCP socket.

- sendsz (input/output)
  The maximum allowable packet size that can be sent. If no size is supplied, a default value is returned.

- `recvsz` (input/output)
  The maximum allowable packet size that can be received. If no size is supplied, a default value is returned.

Explanation

The client uses TCP as the transport protocol. The remote program is located at Internet address `*addr`. If `addr->sin_port` is 0, then it is set to the actual port that the remote program is listening on. (The remote Portmap server is consulted for this information.) The argument `*sockp` is a socket. If it is `RPC_ANYSOCK`, then this function call opens a new socket and sets `*sockp`.

> **Note:** Since TCP-based RPC uses buffered I/O, you can specify the size of the send and receive buffers with the arguments `sendsz` and `recvsz`, respectively. If you specify a value less than 100 for `sendsz` or `recvsz`, the call uses the default size of `3998`.

Return Values

If successful, `clnttcp_create()` returns an RPC handle to a `CLIENT` type definition, as defined in the `clnt.h` include file. If unsuccessful, `clnttcp_create()` returns the value `NULL` and sets the `rpc_createerr` global variable to a value that indicates the type of error.

clntudp_create()

Purpose

The `clntudp_create()` function call creates an RPC client for the remote program `prognum` and version `versnum`.

Format

```
#include <clnt.h>

CLIENT * clntudp_create(addr, prognum, versnum, wait, sockp)

struct sockaddr_in *addr;
unsigned long prognum;
unsigned long versnum;
struct timeval wait;
int *sockp;
```

Arguments

- `addr` (input/output)
  The address of the Internet-style socket address information. This structure is in the `in.h` include file. If the port number is 0, a binder on the remote machine is consulted for the correct port number.

- `prognum` (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range `20000000` to `3fffffff` hexadecimal.

- `versnum` (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of `versnum` should be greater than 0.

- `wait` (input)
  This is the amount of time elapsed between retransmissions of a call when no response was heard after the first call. Retransmission occurs until the actual RPC call times out. The time is defined in the format of the structure `timeval`, which can be found in the `time.h` include file. The time value is expressed in seconds and microseconds.

- `sockp` (input/output)
  A pointer to a requested socket. If the socket value is less than 0, the socket is set to a newly created TCP socket.

Explanation

The client uses UDP as a transport protocol. The remote program is located at Internet address `*addr`. If `addr->sin_port` is 0, then it is set to the actual port on which the remote program is listening. (The remote Portmap server is consulted for this information.) The argument `*sockp` is a socket. If it is `RPC_ANYSOCK`, then this function call opens a new one and sets `*sockp`. UDP resends the call message in intervals of `wait` time until a response is received or until the call times out. The total time for the call to time out is specified by `clnt_call()`.

> **Note:** Since UDP-based RPC messages can only hold up to 4,050 bytes of encoded data, this protocol cannot be used for procedures that take large arguments or return results that would exceed 4,050 bytes.

Return Values

If successful, `clntudp_create()` returns an RPC handle to a `CLIENT` type definition, as defined in the `clnt.h` include file. If unsuccessful, `clntudp_create()` returns the value `NULL` and sets the `rpc_createerr` global variable to a value that indicates the type of error.

get_myaddress()

Purpose

The `get_myaddress()` function call passes the machine's Internet Protocol (IP) address to `*addr` without using the library routines that access the host machine database file. For TCP/IP Version 2, this file is the `machine.db` file.

Format

```
#include <pmap_clnt.h>

bool_t get_myaddress(addr)

struct sockaddr_in *addr;
```

Arguments

- `addr` (output)
  The Internet-style socket address information.

Explanation

The port number is always set to `htons(PMAPPORT)`. If the default K104 adapter is changed between calls to the `get_myaddress()` function call, then the address of the external socket address `rpc_myaddr` should be set to `0.0.0.0.0.0.0.0` before the second call.

pmap_getmaps()

Purpose

The `pmap_getmaps()` function call is a user interface to the Portmap server, which returns a list of the current RPC program-to-port mappings on the host located at Internet Protocol (IP) address `*addr`.

Format

```
#include <pmap_clnt.h>

struct pmaplist * pmap_getmaps(addr)

struct sockaddr_in *addr;
```

Arguments

- `addr` (input/output)
  The address of the Internet-style socket address to be probed for mapping information. This structure is in the `in.h` include file. If the port number is 0, a binder on the remote machine is consulted for the correct port number.

Explanation

The `pmap_getmaps()` function call provides the `rpcinfo -p` command with a list of the current RPC program-to-port mappings, as registered with the host's Portmap server.

Return Values

If successful, `pmap_getmaps()` lists the current RPC program-to-port mappings of the specified host. If unsuccessful, `pmap_getmaps()` returns `NULL`.

pmap_getport()

Purpose

The `pmap_getport()` function call is a user interface to the Portmap server. The function call returns the port number on which a service is waiting. The service is defined by the program number `prognum`, version `versnum`, and the transport protocol associated with `protocol`.

Format

```
#include <pmap_clnt.h>

unsigned short pmap_getport(addr, prognum, versnum, protocol)

struct sockaddr_in *addr;
unsigned long prognum;
unsigned long versnum;
unsigned long protocol;
```

Arguments

- `addr` (input/output)
  The address of the Internet-style socket address to be probed for the requested service. This structure is in the `in.h` include file. If the port number is 0, a binder on the remote machine is consulted for the correct port number.

- `prognum` (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range `20000000` to `3fffffff` hexadecimal.

- `versnum` (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of `versnum` should be greater than 0.

- `protocol` (input)
  The communication protocol requested. The value of `protocol` can be `0`, `IPPROTO_UDP`, or `IPPROTO_TCP`.

Explanation

A return value of 0 means that the mapping does not exist or that the RPC system failed to contact the remote Portmap server. In the latter case, the global variable `rpc_createerr`, which is described in the section "RPC Global Variables" earlier in this chapter, contains the RPC status.

Return Values

If successful, `pmap_getport()` returns the port number for the specified service. If unsuccessful, `pmap_getport()` returns 0.

pmap_rmtcall()

Purpose

The `pmap_rmtcall()` function call is a user interface to the Portmap server, which instructs the Portmap server on the host at Internet Protocol (IP) address `*addr` to make an RPC call to a procedure on that host.

Format

```
#include <pmap_clnt.h>

enum clnt_stat pmap_rmtcall(addr, prognum, versnum, procnum, inproc,
in,
                            outproc, out, tout, portp)

struct sockaddr_in *addr;
unsigned long prognum;
unsigned long versnum;
unsigned long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
unsigned long *portp;
```

Arguments

- `addr` (input/output)
  The address of the Internet-style socket address for the host upon which the remote procedure call is to be made. This structure is in the `in.h` include file. If the port number is 0, a binder on the remote machine is consulted for the correct port number.

- `prognum` (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range `20000000` to `3fffffff` hexadecimal.

- `versnum` (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of `versnum` should be greater than 0.

- `procnum` (input)
  The procedure number within the requested service and version.

- `inproc` (input)
  A pointer to a procedure, as defined by the `xdrproc_t` type definition. This procedure converts the `in` data to XDR format. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The `xdrproc_t` function, defined in the `xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- `in` (input)
  A pointer to character data that contains the procedure's arguments.

- `outproc` (input)
  A pointer to a procedure as defined by the `xdrproc_t` type definition. This procedure converts the data from the result of the call that is in XDR format to host-native format, which is stored in `out`. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The `xdrproc_t` function, defined in the `xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- `out` (output)
  A pointer to character data containing the results of the call.

- `tout` (input)
  A structure containing time-out information in seconds and microseconds, as defined in the `timeval.h` include file.

- `portp` (output)
  Specifies that the return value is the program's port number if the procedure is successful.

Explanation

The `pmap_rmtcall()` function call instructs the Portmap server to call an RPC function for the program that called `pmap_rmtcall()`. The definitions for several of the arguments are discussed in the descriptions of the `callrpc()` and `clnt_call()` function calls. This procedure should be used only as an availability check. See also the description of the `clnt_broadcast()` function call.

pmap_set()

Purpose

The `pmap_set()` function call is a user interface to the Portmap server, which establishes a mapping between the arguments `prognum`, `versnum`, `protocol` and the argument `port` on the machine's Portmap server.

Format

```
#include <pmap_clnt.h>

bool_t pmap_set(prognum, versnum, protocol, port)

unsigned long prognum;
unsigned long versnum;
unsigned long protocol;
unsigned short port;
```

Arguments

- `prognum` (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range `20000000` to `3fffffff` hexadecimal.

- `versnum` (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of `versnum` should be greater than 0.

- `protocol` (input)
  The communication protocol requested. The value of `protocol` can be `0`, `IPPROTO_UDP`, or `IPPROTO_TCP`.

- `port` (input)
  An integer specifying the TCP port of the requested service. The value of `port` must be greater than 0.

Return Values

The `pmap_set()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

pmap_unset()

Purpose

The `pmap_unset()` function call clears the program-number entry from the Portmap server's tables. A program calls `pmap_unset()` to erase any trace of the previous program.

Format

```
#include <pmap_clnt.h>

bool_t pmap_unset(prognum, versnum)

unsigned long prognum;
unsigned long versnum;
```

Arguments

- `prognum` (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range `20000000` to `3fffffff` hexadecimal.

- `versnum` (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of `versnum` should be greater than 0.

Return Values

The `pmap_unset()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

registerrpc()

Purpose

The `registerrpc()` function call, which uses UDP, registers the procedure `procname` with the RPC service package.

Format

```
/* Implicitly defined as int; thus, no include file declaration */

int registerrpc(prognum, versnum, procnum, procname, inproc, outproc)

unsigned long prognum;
unsigned long versnum;
unsigned long procnum;
char *(*procname)();
xdrproc_t inproc;
xdrproc_t outproc;
```

Arguments

- `prognum` (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range `20000000` to `3fffffff` hexadecimal.

- `versnum` (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of `versnum` should be greater than 0.

- `procnum` (input)
  The procedure number within the requested service and version.

- `procname` (input)
  The procedure name to be registered.

- `inproc` (input)
  A pointer to a procedure as defined by the `xdrproc_t` type definition. This procedure converts the `in` data to XDR format. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The `xdrproc_t` function, defined in the `xdr.h` include file, takes two arguments: the first is an XDR handle, the second is a pointer to be converted.

- `outproc` (input)

  A pointer to a procedure as defined by the `xdrproc_t` type definition, which is an XDR routine describing the results in simple primitives. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The function `xdrproc_t`, defined in the `xdr.h` include file, takes two arguments: the first is an XDR handle; the second is a pointer to be converted.

Explanation

The `registerrpc()` function call registers only one procedure at a time. If a request arrives for program `prognum`, version `versnum`, and procedure `procnum`, `procname` is called with a pointer to its argument(s). The `procname` argument returns a pointer to its static result(s). The `inproc` argument decodes the arguments, while the `outproc` argument encodes the result(s). The `registerrpc()` function call uses the `RPC_ANYSOCK` field when creating a transport handle.

> **Note:** Remote procedures registered in this form are accessed by means of UDP. See the description of the `svcudp_create()` function call for restrictions.

Return Values

The `registerrpc()` function call returns 1 if successful and -1 if unsuccessful.

rpc_errmsg()

Purpose

The `rpc_errmsg()` function call returns a pointer to a character string corresponding to the error message returned by an RPC function call such as `callrpc()`.

Format

```
/* Implicitly defined as int; thus, no include file declaration */

char *rpc_errmsg(err)

int err;
```

Arguments

- `err` (input)
  The error number to interpret.

Return Values

The `rpc_errmsg()` function call returns a pointer to a character string corresponding to the error message.

svc_destroy()

Purpose

The `svc_destroy()` function call destroys the RPC service transport handle `xprt`.

Format

```
#include <svc.h>

int svc_destroy(xprt)

SVCXPRT *xprt;
```

Arguments

- `xprt` (input/output)
  The RPC service transport handle. This structure is in the `svc.h` include file, which is located in the directory `>system>rpc_include_library`. Stratus recommends that the program allocate memory for this structure in one of two ways.

  - The program allocates memory for the character data by calling the function `malloc()`.
  - The program passes `NULL` as the value for this argument, which enables an XDR function call to allocate space for the structure.

Explanation

Destruction usually involves deallocation of private data structures, including `xprt` itself. Use of `xprt` is undefined after a call to `svc_destroy()`.

svc_freeargs()

Purpose

The `svc_freeargs()` function call frees any data allocated by the `svc_getargs()` function call when it decoded the arguments of a service procedure. The `svc_freeargs()` function call also frees any memory allocated by the `malloc()` function call.

Format

```
#include <svc.h>

int svc_freeargs(xprt, inproc, in)

SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Arguments

- `xprt` (input)
  The RPC service transport handle. This structure is in the `svc.h` include file, which is located in the directory `>system>rpc_include_library`. Stratus recommends that the program allocate memory for this structure in one of two ways.

  - The program allocates memory for the character data by calling the function `malloc()`.
  - The program passes `NULL` as the value for this argument, which enables an XDR function call to allocate space for the structure.

- `inproc` (input)
  A pointer to a procedure as defined by the `xdrproc_t` type definition. This procedure converts the `in` data to XDR format. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The `xdrproc_t` function, defined in the `xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

- `in` (input)
  A pointer to character data that contains the procedure's arguments.

Return Values

The `svc_freeargs()` function call returns 1 if successful and 0 if unsuccessful.

svc_getargs()

Purpose

The svc_getargs() function call decodes the arguments of an RPC request associated with the RPC service transport handle xprt.

Format

```
#include <svc.h>

int svc_getargs(xprt, inproc, in)

SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Arguments

- xprt (input)
  The RPC service transport handle. This structure is in the svc.h include file, which is located in the directory >system>rpc_include_library. Stratus recommends that the program allocate memory for this structure in one of two ways.
  - The program allocates memory for the character data by calling the function malloc() prior to calling svc_getargs().
  - The program passes NULL as the value for this argument, which enables an XDR function call to allocate space for the structure.
- inproc (input)
  A pointer to a procedure as defined by the xdrproc_t type definition. This procedure converts the in data to XDR format. See, for example, xdr_int(), xdr_long(), xdr_char(), and other XDR function calls. The xdrproc_t function, defined in the xdr.h include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.
- in (input)
  A pointer to character data that contains the procedure's arguments.

Explanation

The svc_getargs() function call passes the arguments to the address in where the XDR routine inproc decodes the arguments.

Return Values

The `svc_getargs()` function call returns 1 if decoding is successful and 0 if unsuccessful.

svc_getcaller()

Purpose

The `svc_getcaller()` function call obtains the network address of the caller of a procedure associated with the RPC service transport handle `xprt`.

Format

```
#include <svc.h>

struct sockaddr_in svc_getcaller(xprt)

SVCXPRT *xprt;
```

Arguments

- `xprt` (input)
  The RPC service transport handle. This structure is in the `svc.h` include file, which is located in the directory `>system>rpc_include_library`. Stratus recommends that the program allocate memory for this structure in one of two ways.

  - The program allocates memory for the character data by calling the function `malloc()`.
  - The program passes `NULL` as the value for this argument, which enables an XDR function call to allocate space for the structure.

svc_getreq()

Purpose

The `svc_getreq()` function call services all of the file descriptors represented in an `rdfds` bit mask by calling the appropriate dispatch routines.

Format

If you are binding an RPC program with the object modules located in the directory `(master_disk)>system>rpc_object_library` (which is the default directory for TCP/IP Version 2), the format of the `svc_getreq()` function call is as follows:

```
#include <svc.h>

void svc_getreq(rdfds)

int rdfds;
```

If you are binding an RPC program with the object modules located in the directory `(master_disk)>system>rpc_tcp_os_object_library` (which is the directory for OS TCP/IP), the format of the `svc_getreq()` function call is as follows:

```
#include <svc.h>

void svc_getreq(rdfds)

fd_set rdfds;
```

Arguments

- `rdfds` (input/output)
  The read file-descriptor bit mask.

Explanation

The `svc_getreq()` function is called if a service implementor does not call `svc_run()` but instead implements custom asynchronous event processing. It is called when the `nselect()` system call of TCP/IP Version 2 determines that an RPC request has arrived at an RPC socket. The resulting read file-descriptor bit mask is `rdfds`. The function call returns when all sockets associated with the value `rdfds` have been serviced.

svc_register()

Purpose

The svc_register() function call associates the prognum and versnum arguments with the service dispatch procedure dispatch().

Format

```
#include <svc.h>

bool_t svc_register(xprt, prognum, versnum, dispatch, protocol)

SVCXPRT *xprt;
unsigned long prognum;
unsigned long versnum;
void (*dispatch)();
int protocol;
```

Arguments

- xprt (input)
  The RPC service transport handle. This structure is in the svc.h include file, which is located in the directory >system>rpc_include_library. Stratus recommends that the program allocate memory for this structure in one of two ways.
  - The program allocates memory for the character data by calling the function malloc().
  - The program passes NULL as the value for this argument, which enables an XDR function call to allocate space for the structure.
- prognum (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range 20000000 to 3fffffff hexadecimal.
- versnum (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of versnum should be greater than 0.
- dispatch (input)
  The dispatch procedure to be associated with program number prognum and version number versnum. (See "Explanation" below.)

- `protocol` (input)
  The communication protocol requested. The value of `protocol` can be `0`, `IPPROTO_UDP`, or `IPPROTO_TCP`.

Explanation

If the value of `protocol` is `0`, the service is not registered with the Portmap server. If the value of `protocol` is nonzero, then a mapping of the triple argument `(prognum, versnum, protocol)` to `xprt->xp_port` is established with the local Portmap server. The `dispatch()` procedure has the following form.

```
dispatch(request, xprt)

struct svc_req *request;
SVCXPRT *xprt;
```

Return Values

The `svc_register()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

svc_run()

Purpose

The `svc_run()` function call waits for RPC requests to arrive. When a request does arrive, `svc_run()` calls the appropriate service procedure using `svc_getreq()`.

Format

```
#include <svc.h>

void svc_run()
```

Explanation

The `svc_run()` function call returns a value of the type `void`. The function call waits for RPC requests to arrive, and when a request does arrive, `svc_run()` calls the appropriate service procedure using `svc_getreq()`. On a module running TCP/IP Version 2, this procedure usually waits for an `nselect()` system call to return.

svc_runable()

Purpose

While a program is executing, the `svc_runable()` function call detects incoming RPC requests.

Format

```
/* Implicitly defined as int; thus, no include file declaration */

int svc_runable()
```

Explanation

When `svc_runable()` detects incoming RPC requests, you process those requests by calling `svc_run()`. If there are no incoming RPC requests, you call `svc_run()` in the background, as shown in the following sample code.

```
for (;;) {
     if (svc_runable()) {
          printf("Incoming request to process\n");
          svc_run();
     }
background_small_portion();
}
```

Return Values

The `svc_runable()` function call returns 1 if there are incoming requests, 0 if no requests have arrived, and -1 if there is an error condition.

svc_sendreply()

Purpose

The svc_sendreply() function call is called by an RPC service dispatch routine to send the results of a remote procedure call.

Format

```
#include <svc.h>

bool_t svc_sendreply(xprt, outproc, out)

SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

Arguments

- xprt (input)
  The RPC service transport handle. This structure is in the svc.h include file, which is located in the directory >system>rpc_include_library. Stratus recommends that the program allocate memory for this structure in one of two ways.
  - The program allocates memory for the character data by calling the function malloc().
  - The program passes NULL as the value for this argument, which enables an XDR function call to allocate space for the structure.
- outproc (input)
  A pointer to a procedure as defined by the xdrproc_t type definition. This procedure converts the data from the result of the call that is in XDR format to host-native format, which is stored in out. See, for example, xdr_int(), xdr_long(), xdr_char(), and other XDR function calls. The xdrproc_t function, defined in the xdr.h include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.
- out (output)
  A pointer to character data containing the results of the call.

Explanation

The argument xprt is the caller's associated transport handle, outproc is the XDR routine used to encode the results, and out is the address of the results.

Return Values

The `svc_sendreply()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

svc_unregister()

Purpose

The `svc_unregister()` function call removes all mappings of the arguments `prognum` and `versnum` to dispatch routines and port numbers.

Format

```
#include <svc.h>

void svc_unregister(prognum, versnum)

unsigned long prognum;
unsigned long versnum;
```

Arguments

- `prognum` (input)
  The RPC program number. This number is determined by the service being called. For a user-created service, the number should be in the range `20000000` to `3fffffff` hexadecimal.

- `versnum` (input)
  The RPC program version. This is the version of the program, or service, the user wants. A program can support multiple versions. The value of `versnum` should be greater than 0.

svcerr_auth()

Purpose

The `svcerr_auth()` function call is called by an RPC service dispatch routine that will not perform a remote procedure call due to an authentication error.

Format

```
#include <svc.h>

void svcerr_auth(xprt, why)

SVCXPRT *xprt;
enum auth_stat why;
```

Arguments

- `xprt` (input)
  The RPC service transport handle. This structure is in the `svc.h` include file, which is located in the directory `>system>rpc_include_library`. Stratus recommends that the program allocate memory for this structure in one of two ways.

  - The program allocates memory for the character data by calling the function `malloc()`.
  - The program passes `NULL` as the value for this argument, which enables an XDR function call to allocate space for the structure.

- `why` (output)
  Specifies the reason for the authentication error. The value of `why` equals the value of the `auth_stat` constant. Table 4-2 lists the status labels and status codes for the `auth_stat` constant.

svcerr_decode()

Purpose

The `svcerr_decode()` function call is called by an RPC service dispatch routine that is unable to decode its arguments. See also the description of the `svc_getargs()` function call.

Format

```
#include <svc.h>

void svcerr_decode(xprt)

SVCXPRT *xprt;
```

Arguments

- `xprt` (input)
  The RPC service transport handle. This structure is in the `svc.h` include file, which is located in the directory `>system>rpc_include_library`. Stratus recommends that the program allocate memory for this structure in one of two ways.

  - The program allocates memory for the character data by calling the function `malloc()`.
  - The program passes `NULL` as the value for this argument, which enables an XDR function call to allocate space for the structure.

svcerr_noproc()

Purpose

The `svcerr_noproc()` function call is called by an RPC service dispatch routine that does not implement the procedure number requested by the caller.

Format

```
#include <svc.h>

void svcerr_noproc(xprt)

SVCXPRT *xprt;
```

Arguments

- `xprt` (input)
  The RPC service transport handle. This structure is in the `svc.h` include file, which is located in the directory `>system>rpc_include_library`. Stratus recommends that the program allocate memory for this structure in one of two ways.

  - The program allocates memory for the character data by calling the function `malloc()`.
  - The program passes `NULL` as the value for this argument, which enables an XDR function call to allocate space for the structure.

svcerr_systemerr()

Purpose

The `svcerr_systemerr()` function call is called by an RPC service dispatch routine when it detects a system error that is not covered by a protocol.

Format

```
#include <svc.h>

void svcerr_systemerr(xprt)

SVCXPRT *xprt;
```

Arguments

- `xprt` (input)
  The RPC service transport handle. This structure is in the `svc.h` include file, which is located in the directory `>system>rpc_include_library`. Stratus recommends that the program allocate memory for this structure in one of two ways.

  - The program allocates memory for the character data by calling the function `malloc()`.
  - The program passes `NULL` as the value for this argument, which enables an XDR function call to allocate space for the structure.

Explanation

If a service can no longer allocate storage, it may call this function call.

svcerr_weakauth()

Purpose

The `svcerr_weakauth()` function call is called by an RPC service dispatch routine that cannot perform a remote procedure call due to insufficient, but correct, authentication arguments.

Format

```
#include <svc.h>

void svcerr_weakauth(xprt)

SVCXPRT *xprt;
```

Arguments

- `xprt` (input)
  The RPC service transport handle. This structure is in the `svc.h` include file, which is located in the directory `>system>rpc_include_library`. Stratus recommends that the program allocate memory for this structure in one of two ways.

  - The program allocates memory for the character data by calling the function `malloc()`.
  - The program passes `NULL` as the value for this argument, which enables an XDR function call to allocate space for the structure.

Explanation

The `svcerr_weakauth()` function calls `svcerr_auth(xprt,AUTH_TOOWEAK)`.

svctcp_create()

Purpose

The `svctcp_create()` function call creates a TCP-based RPC service transport to which it returns a pointer.

Format

```
#include <svc.h>

SVCXPRT * svctcp_create(sock, send_buf_size, recv_buf_size)

int sock;
unsigned int send_buf_size;
unsigned int recv_buf_size;
```

Arguments

- `sock` (input/output)
  A pointer to a requested socket. If the socket value is less than 0, the socket is set to a newly created TCP socket.

- `send_buf_size` (input/output)
  The size of the send buffer. Defaults are returned if the size is 0.

- `recv_buf_size` (input/output)
  The size of the receive buffer. Defaults are returned if the size is 0.

Explanation

The transport is associated with the socket `sock`, which may be `RPC_ANYSOCK`, in which case a new socket is created. If the socket is not bound to a local TCP port, `svctcp_create()` binds it to an arbitrary port. Upon completion, the transport's socket number is `xprt->xp_sock` and the transport's port number is `xprt->xp_port`.

> **Note:** Since TCP-based RPC uses buffered I/O, you can specify the size of the send and receive buffers. If you specify a value of `0` for `send_buf_size` and `recv_buf_size`, the call uses a default size of `0`.

Return Values

If successful, `svctcp_create()` returns an `SVCXPRT` handle, as defined in the `svc.h` include file. If unsuccessful, `svctcp_create()` returns `NULL`.

svcudp_create()

Purpose

The `svcudp_create()` function call creates a UDP-based RPC service transport to which it returns a pointer.

Format

```
#include <svc.h>

SVCXPRT * svcudp_create(sock)

int sock;
```

Arguments

- `sock` (input/output)
  A pointer to a requested socket. If the socket value is less than 0, the socket is set to a newly created UDP socket.

Explanation

The transport is associated with the socket `sock`, which may be `RPC_ANYSOCK`, in which case a new socket is created. If the socket is not bound to a local UDP port, `svcudp_create()` binds it to an arbitrary port. Upon completion, the transport's socket number is `xprt->xp_sock` and the transport's port number is `xprt->xp_port`.

> **Note:** UDP-based RPC messages can hold a maximum of 4,050 bytes of encoded data; therefore, this transport cannot be used for procedures that take large arguments or return results that would exceed 4,050 bytes.

Return Values

If successful, `svcudp_create()` returns an `SVCXPRT` handle, as defined in the `svc.h` include file. If unsuccessful, `svcudp_create()` returns `NULL`.

xdr_array()

Purpose

The `xdr_array()` function call serializes arrays into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)

XDR *xdrs;
char **arrp;
unsigned int *sizep;
unsigned int maxsize;
unsigned int elsize;
xdrproc_t elproc;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `arrp` (input)
  The address of the pointer to the array to be converted.

- `sizep` (input)
  The address of the number of elements in the array. If the value of the `arrp` argument is `NULL`, the space allocated is the value of the `sizep` argument multiplied by `elsize`.

- `maxsize` (input)
  The maximum element count in the array.

- `elsize` (input)
  The size, in bytes, of each element of the array.

- `elproc` (input)
  The XDR procedure to call to handle the conversion of each element in the array.

Explanation

The `arrp` argument is the address of the pointer to the array, while the `sizep` argument is the address of the element count of the array. This element count cannot exceed the `maxsize` argument. The `elsize` argument is the result of the `sizeof()` function of each array

element. The `elproc` argument is an XDR filter that translates between the C form of the array elements and their external representations.

Return Values

The `xdr_array()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_bool()

Purpose

The `xdr_bool()` function call serializes Boolean data into the corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_bool(xdrs, bp)

XDR *xdrs;
bool_t *bp;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `bp` (input/output)
  The address of the Boolean data to be converted.

Explanation

When encoding data, this filter produces the value `0` or `1`.

Return Values

The `xdr_bool()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_bytes()

Purpose

The `xdr_bytes()` function call serializes counted byte strings into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_bytes(xdrs, sp, sizep, maxsize)

XDR *xdrs;
char **sp;
unsigned int *sizep;
unsigned int maxsize;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `sp` (input)
  The address of the string pointer.

- `sizep` (input)
  An address of the number of elements in the array. If the value of the `sp` argument is `NULL`, the space allocated is the value of the `maxsize` argument.

- `maxsize` (input)
  The maximum element count of the array.

Explanation

The `sp` argument is the address of the string pointer. The length of the string is located at the address specified by the `sizep` argument. Strings cannot exceed `maxsize`.

Return Values

The `xdr_bytes()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_double()

Purpose

The `xdr_double()` function call serializes values of the C-language type `double` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_double(xdrs, dp)

XDR *xdrs;
double *dp;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `dp` (input/output)
  The address of the double precision data being converted.

Return Values

The `xdr_double()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_enum()

Purpose

The `xdr_enum()` function call serializes values of the C-language type `enum` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_enum(xdrs, ep)

XDR *xdrs;
enum_t *ep;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `ep` (input/output)
  The address of the enumerated data or integers being converted.

Return Values

The `xdr_enum()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_float()

Purpose

The `xdr_float()` function call serializes values of the C-language type `float` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_float(xdrs, fp)

XDR *xdrs;
float *fp;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `fp` (input/output)
  The address of the floating-point data being converted.

Return Values

The `xdr_float()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_int()

Purpose

The `xdr_int()` function call serializes values of the C-language type `int` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_int(xdrs, ip)

XDR *xdrs;
int *ip;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `ip` (input/output)
  The address of the integer being converted.

Return Values

The `xdr_int()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_long()

Purpose

The `xdr_long()` function call serializes values of the C-language type `long` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_long(xdrs, lp)

XDR *xdrs;
long *lp;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `lp` (input/output)
  The address of the long integer being converted.

Return Values

The `xdr_long()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_opaque()

Purpose

The `xdr_opaque()` function call serializes fixed-size opaque data into their corresponding external representations, and deserializes those external representations into the fixed-size opaque data.

Format

```
#include <xdr.h>

bool_t xdr_opaque(xdrs, cp, cnt)

XDR *xdrs;
char *cp;
unsigned int cnt;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `cp` (input/output)
  The address of the opaque data to be converted.

- `cnt` (input/output)
  The size, in bytes, of the opaque data to be converted.

Return Values

The `xdr_opaque()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_reference()

Purpose

The `xdr_reference()` function call provides pointer chasing within structures.

Format

```
#include <xdr.h>

bool_t xdr_reference(xdrs, pp, size, proc)

XDR *xdrs;
char **pp;
unsigned int size;
xdrproc_t proc;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `pp` (input/output)
  The address of a pointer to storage. If the value is `NULL`, the necessary storage is allocated.

- `size` (input/output)
  The size of the referenced structure.

- `proc` (input/output)
  The XDR procedure that filters the structure between its C form and its external representation.

Explanation

The `pp` argument is the address of the pointer. The `size` argument is the size of the structure, specified in the `sizeof()` function call, to which the `pp` argument points.

Return Values

The `xdr_reference()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_short()

Purpose

The `xdr_short()` function call serializes values of the C-language type `short` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_short(xdrs, sp)

XDR *xdrs;
short *sp;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `sp` (input/output)
  The address of the short integer being converted.

Return Values

The `xdr_short()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_string()

Purpose

The `xdr_string()` function call serializes a string from its C-language representation into its corresponding external representation, and deserializes that external representation into the C-language representation.

Format

```
#include <xdr.h>

bool_t xdr_string(xdrs, sp, maxsize)

XDR *xdrs;
char **sp;
unsigned int maxsize;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `sp` (input)
  The address of the pointer to the string to be converted.

- `maxsize` (input)
  The maximum length of the string.

Explanation

The length of the C strings cannot exceed the number specified in the `maxsize` argument. The `sp` argument is the address of the pointer to the string.

Return Values

The `xdr_string()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_u_int()

Purpose

The `xdr_u_int()` function call serializes values of the C-language type `unsigned` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_u_int(xdrs, up)

XDR *xdrs;
unsigned int *up;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `up` (input/output)
  The address of the unsigned integer being converted.

Return Values

The `xdr_u_int()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_u_long()

Purpose

The `xdr_u_long()` function call serializes values of the C-language type `unsigned long` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_u_long(xdrs, ulp)

XDR *xdrs;
unsigned long *ulp;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `ulp` (input/output)
  The address of the unsigned long integer being converted.

Return Values

The `xdr_u_long()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_u_short()

Purpose

The `xdr_u_short()` function call serializes values of the C-language type `unsigned short` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_u_short(xdrs, usp)

XDR *xdrs;
unsigned short *usp;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `usp` (input/output)
  The address of the unsigned short integer being converted.

Return Values

The `xdr_u_short()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_union()

Purpose

The `xdr_union()` function call serializes values of the C-language type `union` into their corresponding external representations, and deserializes those external representations into their C-language representations.

Format

```
#include <xdr.h>

bool_t xdr_union(xdrs, dscmp, unp, choices, dfault)

XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `dscmp` (input)
  The address of the union's discriminant value.

- `unp` (output)
  The address of the union itself.

- `choices` (input)
  The address of an array of `xdr_discrim` structures. The structure consists of an integer value and a procedure to handle the associated part of the union. It is terminated with an entry containing a `NULL` procedure pointer.

- `dfault` (input)
  A default procedure called if there is no specified routine as defined by the `xdrproc_t` type definition. The `xdrproc_t` function converts the data from the result of the call that is in XDR format to host-native format, which is stored in the address of the union `unp`. See, for example, `xdr_int()`, `xdr_long()`, `xdr_char()`, and other XDR function calls. The `xdrproc_t` function, defined in the `xdr.h` include file, takes two arguments. The first is an XDR handle; the second is a pointer to be converted.

Explanation

The `dscmp` argument is the address of the union's discriminant value, while the `unp` argument is the address of the union itself.

Return Values

The `xdr_union()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

xdr_void()

Purpose

The `xdr_void()` function call converts a void result.

Format

```
#include <xdr.h>

bool_t xdr_void()
```

Return Values

The `xdr_void()` function call always returns 1.

xdr_wrapstring()

Purpose

The `xdr_wrapstring()` function calls `xdr_string(xdrs, sp, MAXUNSIGNED)`, where `MAXUNSIGNED` is the maximum value of an unsigned integer.

Format

```
#include <xdr.h>

bool_t xdr_wrapstring(xdrs, sp)

XDR *xdrs;
char **sp;
```

Arguments

- `xdrs` (input)
  The XDR handle. This structure is in the `xdr.h` include file.

- `sp` (input/output)
  The address of a pointer to the string to be converted.

Explanation

This function call is useful because the RPC package requires only two arguments, whereas `xdr_string()`, one of the most frequently used primitives, requires three arguments.

Return Values

The `xdr_wrapstring()` function call returns 1 if successful and 0 if unsuccessful. The return value is of the type `bool_t`, which is defined as an `int` in the `types.h` include file.

Chapter 5:
Compiling, Binding, and Debugging RPC Programs

This chapter describes how to compile and bind RPC programs. It identifies the directories that should be in your library search paths and the include files that should be part of your programs. This chapter also describes commands that are helpful in debugging RPC programs.

> **Caution: If you intend to convert the module on which you execute RPC programs from TCP/IP Version 2 to OS TCP/IP, you must be familiar with the information in the file** `os_tcp_port_guide.doc`, which is located in the directory `(master_disk)>system>doc>tcp_os`. This file, entitled *Porting Guide: TCP/IP 2.0 to OS TCP/IP*, explains how to port programs from TCP/IP Version 2 to OS TCP/IP.

Compiling and Binding RPC Programs

Any program that uses RPC function calls **must** contain an `#include` statement for the include file `tcp_socket.h`, which contains TCP/IP and socket definitions. The statement `#include <tcp_socket.h>` must appear in the program **before** `#include` statements for any RPC include files. For TCP/IP Version 2, the file `tcp_socket.h` is located in the directory `(master_disk)>system>tcp/ip_include_library`. For OS TCP/IP, this file is located in the directory `(master_disk)>system>tcp_os>include_library`. You should ensure that the appropriate include library is in the list of include-library search paths.

The program must also contain `#include` statements for some or all of the following RPC include files.

- `auth.h`
- `auth_unix.h`
- `clnt.h`
- `if.h`
- `In.h`
- `Netdb.h`
- `old_in.h`
- `old_netdb.h`
- `pmap_clnt.h`
- `pmap_prot.h`
- `rpc.h`
- `rpc_errno.h`
- `rpc_macros.h`

- `rpc_msg.h`
- `rpcrealtime.h`
- `rpctypes.h`
- `Stat.h`
- `svc.h`
- `svc_auth.h`
- `tcp_rtn_defs.h`
- `Time.h`
- `Types.h`
- `types.h`
- `xdr.h`

These include files contain definitions of structures that are used with the RPC function calls. RPC include files are located in the directory `(master_disk)>system>rpc_include_library`, which should be in your include-library search paths.

After compiling, you must bind RPC programs with object modules located in one of two directories. If your module is running OS TCP/IP, you **must** bind RPC programs with the object modules located in the directory `(master_disk)>system>rpc_tcp_os_object_library` rather than the default directory `(master_disk)>system>rpc_object_library`. You should ensure that your object-library search paths include this directory. (If your module is running TCP/IP Version 2, you bind RPC programs with the object modules located in the directory `(master_disk)>system>rpc_object_library`, which is the default object-library search path.)

In addition, you must bind RPC programs with the TCP/IP run-time object module and list the object module in the bind control file. For TCP/IP Version 2, the run-time object module is located in the file `(master_disk)>system>tcp/ip_object_library>tcp_runtime.obj`.

To run an RPC program on OpenVOS, the program must be bound with the OpenVOS kernel.

Debugging RPC Programs

The OpenVOS operating system offers several tools that can help you debug RPC programs.

- The `debug` command calls the OpenVOS debugger, which allows you to run your programs in a controlled fashion. Using the debugger, you can set break points, look at the contents of memory locations and registers, and run the program one step at a time. Because some of the RPC and TCP/IP function calls are real-time routines, the `debug` command may be particularly useful. See the *OpenVOS Commands Reference Manual* (R098) for a description of the `debug` command.

- The `mp_debug` command calls the multiprocess debugger to debug one or more processes that can be running anywhere in your network. This command is particularly useful if you want to debug a process that does not typically have a terminal associated with it, such as a server process. See the *OpenVOS Commands Reference Manual* (R098) for a description of the `mp_debug` command.

- The `rpcinfo` command displays information about the RPC services running on an RPC host, including the program numbers, version numbers, protocols, ports, and service names. For more information on the `rpcinfo` command, see ‘‘Monitoring RPC’’ on page 2-3.

- The `tcp_admin` command allows you to monitor one or more K104 Ethernet Communications I/O Adapters running TCP/IP Version 2, as well as monitor programs that have opened a socket on one of these adapters. For detailed information about the `tcp_admin` command of TCP/IP Version 2, see the manual *VOS Communications Software: TCP/IP Administration* (R196).

- The Stratus Ethernet Packet Monitor utility allows you to interactively monitor and display data that is sent and received by a K104 adapter running TCP/IP Version 2. For each packet sent or received by a K104 adapter, the Packet Monitor utility displays both header information and data contained in the packet. For detailed information about the Packet Monitor utility of TCP/IP Version 2, see the manual *VOS Communications Software: TCP/IP Administration* (R196).

Appendix A:
Sample Programs

This appendix presents two sample C programs, a server program and a client program, that use RPC and XDR function calls to broadcast system messages. Because these programs run with OS TCP/IP, you must bind them with the following OS TCP/IP object modules, which are located in the directory `(master_disk)>system>tcp_os>object_library`.

- `tcp_runtime.obj`
- `res_send.obj`
- `tcp_gethost.obj`

The Server Program

```
/* The server example, sample.server_side.c */

#include <new_stdio.h>
#include <rpc.h>
#include <msg_svc.h>
#include <svc.h>
#include <socket.h>

void     exit();
int      set_device();
int      setsockopt();

main(argc, argv)
     int   argc;
     char  **argv;
{
     SVCXPRT      *transp;
     int           ack_and_reply_bcast();

     if (argc != 3) {
        printf("USAGE: server_side tcp_adapter adapter_type[tcp_os,
tcp_ip]\n");
          exit(-1);
     }
```

(Continued on next page)

```
/* Explicitly set the server's default adapter. */

     if (set_device(argv[1]) < 0) {
            printf("Server: Could not set default adapter to %s\n",
argv[1]);
             exit(-1);
     }

/* Initialize with the call svcudp_create() and assign a transport
handle,
   transp, where the program will be registered. */

     transp = svcudp_create(RPC_ANYSOCK);
     if (transp == NULL) {
          printf("can't register CLIENT_SIDE service\n");
          exit(1);
     }

     if (!strcmp("tcp_os", argv[2])) {

/* Set broadcast option for this socket when using OS TCP/IP. */

        setsockopt(transp->xp_sock, SOL_SOCKET, SO_BROADCAST, TRUE,
0);
     }

/* Destroy any previous assignments of values for MESSAGEPROG
(prognum)
   and MESSAGEVERS (versnum) with pmap_unset(). */

     pmap_unset(MESSAGEPROG, MESSAGEVERS);

/* Register the program with the portmapper by calling
svc_register(), which
  also assigns a program number and version number to the program. */

     if (!svc_register(transp, MESSAGEPROG, MESSAGEVERS,
                          ack_and_reply_bcast, IPPROTO_UDP)) {
          printf("can't register CLIENT_SIDE service\n");
          exit(1);
     }
/* Run the call svc_run(), which continuously calls svc_getreq(),
   waiting for client requests. */

     svc_run();     /* never returns */
     printf("Should NEVER reach this point!\n");

/* If, for whatever reason, the call svc_run() returns unexpectedly,
  unregister the service program and destroy the transport handle. */
```

(Continued on next page)

```
     svc_unregister(MESSAGEPROG, MESSAGEVERS);
     svc_destroy(transp);
}

/* The server waits for a request from the portmapper.  The portmapper
sends
   the server a request, which contains a program number, version
number, and
  an address of the client.  Then the server sends an svc_sendreply()
call
   directly to the client.  */

ack_and_reply_bcast(rqstp, transp)
     struct svc_req     *rqstp;
     SVCXPRT          *transp;
{
     unsigned long     u_long_reply;

     switch (rqstp->rq_proc) {

/* In the svc_sendreply() call, the server returns to the client a
value
   indicating the results of the clnt_call() call. */

          case NULLPROC:
               if (!svc_sendreply(transp, xdr_void, 0))
                    printf("Can't reply to RPC call\n");
               svc_freeargs(transp, xdr_void, 0);
          case BROADCAST_MESSAGE:
               u_long_reply = 1;
               printf("RECEIVED: %s\n", "clnt_broadcast");
              if (!svc_sendreply(transp, xdr_u_long, &u_long_reply))
                    printf("Can't reply to RPC call\n");
               svc_freeargs(transp, xdr_u_long, &u_long_reply);
          case CALL_MESSAGE:
               u_long_reply = 1;
               printf("RECEIVED: %s\n", "clnt_call");
              if (!svc_sendreply(transp, xdr_u_long, &u_long_reply))
                    printf("Can't reply to RPC call\n");
               svc_freeargs(transp, xdr_u_long; &u_long_reply);
          default:
               svcerr_noproc(transp);
     }
     xprt_unregister(transp);
     return;
}
```

The Client Program

```
/* The client example, sample.client_side.c */

#include <stdio.h>
#include <tcp_socket.h>
#include "msg_clnt.h"     /* msg.h will be generated by rpcgen. */

#define size_t ANSI_SIZE_T
#include <rpc.h>              /* Always needed */
#include <rpc_macros.h>
#include <time.h>
#include <netdb.h>
#include <pmap_clnt.h>
#include <string.h>
#undef size_t

void           exit();
struct hostent gethostbyname();
int            set_device();
extern int     net_close();

main(argc, argv)
     int      argc;
     char **argv;
{
     CLIENT           *cl;
     struct hostent *hp;
     int                *result;
     char           *server;
     char           *message;
     struct timeval     wait;
     int               sock;
     struct in_sockaddr addr;

     if (argc != 3) {
          printf("USAGE: client_side hostname tcp_adapter\n");
          exit(-1);
     }

/* Explicitly set the client's default adapter. */

     if (set_device(argv[2]) < 0) {
            printf("Client: Could not set default adapter to %s\n",
argv[2]);
             exit(-1);
     }

     if ((hp = gethhostbyname(argv[1], AF_INET)) == NULL) {
          printf("Client: can't get addr for %s\n", argv[1]);
          exit(-1);
     }
```

(Continued on next page)

```
     memcpy((caddr_t)&addr.sin_addr, hp->h_addr, hp->h_length);

     cl      = NULL;
     server  = "no host";
     message = "Stratus Broadcast Message";

     wait.tv_sec      = 5;
     wait.tv_usec     = 0;
     sock             = RPC_ANYSOCK;
     addr.sin_family  = AF_INET;
     addr.sin_port    = 0;

/* Initialize with clntudp_create().  Assign a client handle, clnt,
which is used
   to call MESSAGEPROG on the server specified in the command line.
The
  clntudp_create() function call specifies that RPC will use UDP when
contacting
   the server. */

     cl = clntudp_create(&addr, MESSAGEPROG, MESSAGEVERS, wait,
&sock);

/* Print the error message, if any, stored in the global variable
   rpc_createerr. */

     if (cl == NULL) {
          clnt_pcreateerror(server);
          exit(1);
     }

 /* Call the remote procedure "printmessage" on the server. */

     result = printmessage_1(&message, cl);
     if (result == NULL) {
          /*  An error occurred while calling the server.
           *  Print error message and die.
           */
          clnt_perror(cl, server);
          goto FINISH;
     }

/* If the remote procedure call succeeded, check that the procedure
executed
   properly.  */

     if (*result == 0) {
          printf("Broadcast: %s couldn't print your message\n",
                server);
          goto FINISH;
     }
```

(Continued on next page)

```
/* If the message was printed on the server's console, print the next
   message. */

     printf("Message delivered to %s!\n", server);

/* Clean up by destroying the client handle and closing the socket
before
   exiting. */

FINISH;
     clnt_destroy(cl);
     net_close(sock);
     exit(1);
}

bool_t eachresult( resultsp, raddr )

int  *resultsp;
struct sockaddr_in *raddr;

{
     int count;

     count++;
     printf(stderr, "Broadcast received response %d\n", count );

     if (count == 2)
          return( TRUE );

     return( FALSE );
}

int *
printmessage_1(argp, clnt)
     char **argp;
     CLIENT *clnt;
{
     enum clnt_stat     status;
     static int res;
     int     rpc_status;
     struct timeval     timout;

     bzero((char *)&res, sizeof(res));

     timout.tv_sec = 25;
     timout.tv_usec = 0;
    rpc_status = clnt_call(clnt, PRINTMESSAGE, xdr_wrapstring, argp,
xdr_int,
                                &res, timout);
```

(Continued on next page)

```
     if (rpc_status != RPC_SUCCESS) {
          clnt_perror(clnt, "rpc");
          exit(-1);
     } else if (( status = clnt_broadcast(MESSAGEPROG, MESSAGEVERS,
                         PRINTMESSAGE, xdr_wrapstring,
                         argp, xdr_int, &res,
                         eachresult)) != RPC_SUCCESS) {
               return (NULL);
            }
     return (&res);
}
```

Glossary

access

To read from or write to a file or device. See **access rights**.

access code

A code used in OpenVOS access control lists (ACLs) and default access control lists to show access rights. The following access rights and codes pertain to files.

Execute	e
Null	n
Read	r
Write	w

The following access rights and codes pertain to directories.

Modify	m
Null	n
Status	s
Undefined	n

access control

The mechanism that OpenVOS uses to determine a user's access rights to files and directories.

access control list (ACL)

A list that OpenVOS uses to determine a user's access rights to a particular file or directory. An ACL is a list of entries, each of which shows an access code and a user name. An example of an ACL for a directory follows.

```
m    Jones.sales
s    *.sales
n    *.*
```

An example of an ACL for a file follows.

```
w    Jones.sales
r    *.sales
n    *.*
```

access rights

A OpenVOS designation that determines the operations that a user can perform on a file or directory. The types of access rights to a file are null, execute, read, and write. The types of access rights to a directory are null, status, and modify.

ACL

See **access control list (ACL)**.

address

A name that specifies a particular location or machine on a network or group of networks.

application

A program or set of programs that runs on a client.

application process

The user program. In the application layer of the OSI model, an application process is a collection of elements required for information processing.

argument

A character string that specifies how a command, request, subroutine, or function is to be executed.

authentication

The validation of a user's credentials.

bind

To combine a set of one or more independently compiled object modules into a program module. (The UNIX term "load" is equivalent to "bind.") Binding compacts the code and resolves symbolic references to external programs and variables that are shared by object modules in the set and in the object library (see **library**).

bit

The smallest unit of internal computer storage. A bit has one of two values, `0` or `1`.

bound socket

Refers to a socket whose destination address, port, and machine address have already been selected.

buffer

A space reserved in computer memory for temporarily storing data, usually just before transmitting it or just after receiving it.

byte

Eight bits of data. An unsigned byte variable can contain integer values in the range 0 to 255; a signed byte variable can contain integer values in the range -128 to 127.

CAC

See **Customer Assistance Center (CAC)**.

client

1. A user who accesses files on your system while logged in to another system.
2. A process (in general, a user process) that requests services from a module other than the one on which it is executing.

client user

A user who accesses files on your system while logged in to a client system.

compiler

A program that translates a source module (source code) into machine code. The generated machine code is stored in an object module.

configuration table

A table file that OpenVOS uses to identify the elements of a system.

credentials

In UNIX terminology, credentials consist of user and group IDs. In the RPC environment, credentials may also include the host's Internet address.

Customer Assistance Center (CAC)

The central point in a Remote Service Network (RSN). The Customer Assistance Center is sometimes referred to as the Hub.

datagram

A self-contained data packet with a complete address, which can therefore be routed from source to destination without relying on earlier exchanges between the source or destination and the transporting network.

debugger

A OpenVOS tool used as an aid in finding program errors.

default value

The value or attribute used when a necessary value or attribute is omitted.

deserialize

To decode a message into its component data structures or items.

directory

A segment of disk storage that contains files, links, and subdirectories and has its own access limitations.

directory hierarchy

The structure of the set of directories on a disk.

discriminated union

A structure consisting of multiple overlaying records in which an initial element is used to select the record type.

disk directory

The top directory on a disk. In VOS, the disk directory name is prefixed with a number sign (#).

In UNIX, the disk directory is called the *root*.

distributed

Refers to applications that are spread among multiple processes and/or processors (possibly multiple machines) that share the load of an application.

error code

An arithmetic value (usually, a two-byte integer) indicating what, if any, error has occurred (usually, a OpenVOS status code). An error-code argument is often included in subroutines.

error message

A character string that is associated with an error code.

Ethernet

A local area network based on the specifications published by Digital Equipment Corp., Xerox, and Intel. It is a baseband communications system employing a bus topology. IEEE 802.3 defines Carrier Sense Multiple Access/Collision Detection (CSMA/CD) as the access control method for Ethernet. Ethernet cables can be extended to a maximum length of 1,500 meters.

Ethernet adapter

A K104 Ethernet Communications I/O Adapter on which the Stratus Ethernet software runs.

export

To place on public notice the availability of a resource or service. The mechanics of exporting can vary, based on the type of object. For example, exporting a file system involves adding an entry to a control file; exporting a subroutine involves defining the subroutine as global or external.

External Data Representation (XDR)

A means of transporting data between machines of different architectures so that problems, such as byte ordering of integer binary data, are transparent to the application.

file system

A group of directories and files considered as a unit.

In UNIX, a file system is associated with a logical device.

file system data

The text or contents of operating system data files, executable files, or image files.

file system operations

Activities such as reading, writing, creating, deleting, and renaming files, creating and deleting directories, and getting and setting attributes.

fixed file

A file with a fixed organization. In a fixed file, all records are the same size. Each record is stored in a disk or tape region holding a number of bytes that is the same for all the records in the file. Compare with **relative file**, **sequential file**, and **stream file**.

handle

A unique identifier, usually a pointer or an integer value that is passed to subroutines. In OpenVOS, a port ID is an example of a handle.

hexadecimal

Base 16.

host

In the context of networking, any processor attached to and accessed from a network.

include file

A file that the compiler includes in the source module used by the compilation process. The name of the include file must be specified in a language-specific directive within the source module.

include library

A directory that OpenVOS searches for include files.

Internet

A group of interconnected networks, also referred to as the Defense Data Network (DDN)/Defense Advanced Research Projects Agency (DARPA) Internet. The DDN/DARPA Internet is administered by the Network Information Center (NIC), which assigns IP network numbers (addresses) to networks and also registers networks, hosts, and domains on the DDN/DARPA Internet.

Internet Protocol (IP)

A protocol that enables transmission of blocks of data from sources to destinations throughout the Internet. IP provides services to transport-layer protocols (such as TCP, UDP, and FTP) and relies on the services of lower network-layer protocols.

library

One or more directories in which OpenVOS looks for objects of a particular type. There are four types of libraries defined by OpenVOS:

- include libraries, in which the compilers search for include files
- object libraries, in which the binder searches for object modules
- command libraries, in which the command processor searches for commands
- message libraries, in which the operating system searches for message files associated with `.pm` files.

Each library is available in the `>system` directory of each module for all processes running on the module. In addition, you can define your own libraries.

modify access

A type of OpenVOS directory access that gives a user full access to the contents of the directory, including the ability to create, delete, and rename objects.

module star name

A name that contains one or more asterisks or consists solely of an asterisk, used to specify a set of modules in a Stratus system. An asterisk can be in any position in the name, and each asterisk represents zero or more characters. A module star name can be used alone or as part of a full module path name; however, in a full module path name, there can be no asterisks in the system name. When a module star name consists solely of an asterisk, it represents all modules in the current system. See also **star name**.

`module_start_up.cm` **file**

A command file that OpenVOS reads when starting up a module.

mount point

The path name of a particular directory in a directory hierarchy. The user can access that directory and its subdirectories as a file system.

multithreading

Refers to programs that can perform multiple tasks concurrently.

network

A communications facility that connects two or more points. The Stratus network structures can consist of both local networks and long-haul networks.

networkservices

The means of providing functionality across a network. For example, the print service enables one machine to print on another machine's printer.

null access

A type of OpenVOS file or directory access that denies a user access to a file or directory.

object library

A directory that OpenVOS searches for object modules.

opaque

Refers to data items that pass through unconverted and uninterpreted.

optional argument

A command argument for which the operating system does not need a value in order to execute the command.

path name

A unique name that identifies a device or locates an object in the directory hierarchy.

portable

1. Facilitating the movement of code, programs, or applications between machine types and/or operating systems.
2. Machine or operating-system independent.

primitives

The lowest-level or most basic elements upon which functions, languages, operating systems, applications, and libraries are based.

privileged process

An attribute of a user that allows that user to use certain commands, requests, and subroutines. A user can be privileged or not, depending on the user's status as defined in the registration databases and on how the user logged in.

procedure call

A call from a program to an executable procedure. Once the procedure has finished executing, control returns to the original program through a saved address in the calling program.

program

One or more procedures, from one or more source modules, that perform a specific task.

protocol

A specification for the format and relative timing of information exchanged between communicating devices or systems.

read access

A type of OpenVOS file access that allows a user to read the file or execute it (if it is executable), but not write it.

relative file

A file with a relative organization. In a relative file, the records can be different sizes. Each record is stored in a disk or tape region holding a number of bytes that is the same for all the records in the file. Compare with **fixed file**, **sequential file**, and **stream file**.

Remote Procedure Call (RPC) facility

A facility that enables communication with remote services in a way similar to the procedure-calling mechanism available in many programming languages. The RPC facility consists of a library of function calls and a specification for portable data transmission, known as External Data Representation (XDR). Both RPC and XDR are portable, providing a standard I/O library for interprocess communication.

Remote Service Network (RSN)

A facility that connects a Stratus system to the Customer Assistance Center (CAC) through a modem. The RSN automatically reports many hardware and software failures to the CAC.

required argument

A command argument for which you must specify a value.

root

1. The top of a directory hierarchy on a particular device.

2. On the UNIX operating system, a user who has all privileges and has access to all of the data. See **superuser**.

RPC

See **Remote Procedure Call (RPC) facility**.

RSN

See **Remote Service Network (RSN)**.

sequential file

A file with a sequential organization. In a sequential file, the records can be different sizes, and each record is stored in a disk or tape region holding approximately the same number of bytes as in the record. Thus, the record-storage regions in a sequential file vary from record to record. Compare with **fixed file**, **relative file**, and **stream file**.

serialize

To encode data structures and data items into a message.

server

A system process whose purpose is to receive and respond to requests from clients.

service

A facility supplied by a server.

socket

1. A virtual connection point on a module that is used for network communications.
2. In a TCP/IP virtual circuit, communications endpoints to which addresses (names) must be bound. A TCP/IP virtual circuit connection is always between two sockets.

star name

A name that contains one or more asterisks or consists solely of an asterisk. A star name can be used to specify a set of objects. Star names function in the following manner.

- An asterisk can be in any position in a star name.
- In a path name, a star name can be in the final name position only.
- When the operating system matches non-star names to a star name, each asterisk represents zero or more characters.
- A name cannot contain consecutive asterisks; there must always be an intervening character.

Some names that contain asterisks function differently; see **module star name** and **user star name**.

status access

A type of OpenVOS directory access that allows a user to display information about the directory, but not modify the directory by creating, deleting, or renaming objects.

stream file

A file with a sequential organization. In a stream file, the records can be different sizes, and each record is stored in a disk or tape region holding approximately the same number of bytes as in the record. Thus, the record-storage regions in a stream file vary from record to record. In these ways, stream files are similar to sequential files; however, stream files differ from sequential files in the following ways.

1. While a sequential file must be accessed on a record basis, a stream file can be accessed on either a record or byte basis. For example, to read from a sequential file, you must use the OpenVOS subroutine `s$seq_read`, which reads the next record in the file. To read from a stream file, you can use either `s$seq_read` to read the next record, or `s$read_raw` to read a specified number of bytes from the file, ignoring the file's record structure.

2. Sequential files are stored on disk with the record size at the beginning and end of each record. Stream files do not have any record-size information stored with them; each new-line character in the file is interpreted as the end of a record.

   When stream files are used to store text, each record contains one line of text.

   Compare with **fixed file**, **relative file**, and **sequential file**.

subsystem

A OpenVOS facility that enters a command loop in which you can issue directives or requests that have functions unique to the subsystem. The most common subsystems are analyze-system, system-operator, and the debugger.

superuser

A user on the UNIX operating system, typically called `root`, who has all privileges and has access to all of the data.

The user ID for a superuser is `uid 0`.

system primitives

The base subroutine calls and interfaces provided by an operating system.

TCP

See **Transmission Control Protocol (TCP)**.

TCP/IP

See **Transmission Control Protocol/Internet Protocol (TCP/IP)**.

thread

A path of execution with its own unique processor state and stack; similar to a task.

Transmission Control Protocol (TCP)

A transport-level virtual circuit protocol that is implemented over the Internet Protocol (IP) network layer. See **Internet Protocol (IP).**

Transmission Control Protocol/Internet Protocol (TCP/IP)

A suite of protocols that have been a Department of Defense standard since 1968 and were officially adopted on an ARPANET-wide basis in 1983. TCP/IP is used generically to refer to a suite of protocols that include TCP, UDP, IP, ARP, RARP, Telnet, and FTP.

transparency

The interoperability between two applications running on different machines, operating systems, or networks, that enables the applications to have the same functional and operational attributes.

UDP

See **User Datagram Protocol (UDP)**.

unbound socket

Refers to a socket whose destination address, port, and machine address have not yet been selected.

user

1. A person who is registered to use a system. A OpenVOS user is specified by a user name, which consists of a person name and a group name.

2. A person who is logged in on a client machine.

User Datagram Protocol (UDP)

An unreliable datagram-based protocol implemented under TCP/IP that allows two-way message transmission.

user star name

A user name containing one or two asterisks that is used to specify a set of users. When a user attempts to use a file or directory to which an access control list (ACL) applies, the operating system checks the user's access by matching user star names on the list to actual users and groups.

Either component of a user star name (the person name or the group name) can be an asterisk, or both components can be asterisks. An asterisk as the first component matches all person names; an asterisk as the second component matches all group names. In arguments that accept user star names, if only a person name (or only a single asterisk) is specified, the operating system appends `.*` to the name.

XDR

See **External Data Representation (XDR).**

Index

E

F

G

I

K

L

M

N

O

P

T

U

V

X