Network Communications Guide

IRIS-4D Series

SiliconGraphics
Computer Systems

Document number: 007-0810-010

Network Communications Guide

Document Version 1.0

Document Number 007-0810-010

11/90

Technical Publications:

Wendy Ferguson
Beth Fryer
Claudia Lohnes

Engineering:

Andrew Cherenson
Vernon Schryver

This document uses material from chapters of the 4.3BSD System Manager's Manual and Programmer's Supplementary Documents, from Sun Microsystems NFS documentation, and from various Internet Request For Comments documents.

Network Communications Guide
Document Version 1.0
Document Number 007-0810-010

Silicon Graphics, Inc.
Mountain View, California

Contents

List of Figures

List of Tables

1. Communications Overview

This guide is designed to help you learn to use, configure, and manage the network communication software that runs on your Silicon Graphics, Inc., computer, e.g., an IRIS-4D™, Personal IRIS™, or Power Series™. The information that is supplied is also for programmers who want to design and write programs to implement the network communication software. This software is derived from the networking software in the 4.3BSD UNIX release from the University of California at Berkeley and the Sun Microsystems RPC system.

Note for network programmers and administrators: the IRIX® operating system implements the Internet protocol suite and UNIX domain sockets using the 4.3BSD UNIX socket mechanism. The system also supports access to the underlying network media using raw sockets. IRIX does not support the Xerox NS protocol suite.

This guide does not describe how to physically connect your computer to the network; the *Owner's Guide* for your particular system explains this connection.

1.1 Using This Guide

This guide is written for three types of users:

- a person who wants to learn about the communications facilities available under the IRIX operating system and/or who wants to take advantage of the network utilities to communicate with remote machines
- a network programmer
- a network administrator

If you are new to the world of networking, you may want to refer to your *Owner's Guide*, which describes how to use the network via the WorkSpace Transfer Manager, an easy-to-use visual interface to the networking utilities. Otherwise, continue with the rest of this chapter, and then read Chapter 2.

If you are learning about network administration and are responsible for setting up and maintaining the network, you may want to refer to your *Owner's Guide*, which describes how to set up and maintain the network via the WorkSpace Network Manager, an easy-to-use visual interface to the network.

There are three main sections in this guide. The following table lists the section and chapters to see for your particular area of interest.

To find out about:	See section and chapters:
Communication overview	Part 1: 1
Network utilities	Part 1: 1, 2
Network programming	Part 2: 3, 4, 5, 6, 7
RPC, XDR, *rpcgen*	Part 2: 4, 5, 6, 7
Set up the network and maintain it	Part 3: 8, 9
Set up mail	Part 3: Appendix A

Table 1-1. Sections and Chapters in This Guide

1.2 Getting Started

This document makes some assumptions about you and your computer:

- For Part 1, you know a little bit about the UNIX operating system, can enter commands from the system prompt, and are familiar with a text editor such as *vi*. If not, see the *Owner's Guide* for your particular system.
- For Parts 2 and 3, you are a programmer/network administrator, are familiar with the C programming language, and have a working knowledge of network theory and the UNIX operating system.

1.3 Chapter Summary

This guide consolidates all of the material from the former *Communications Guide* and *TCP/IP User's Guide*, and incorporates parts of the *NFS User's Guide*, the Berkeley 4.3BSD *System Manager's Manual*, and various Internet Request for Comment documents.

The following paragraphs briefly describe the content of each chapter. See "Documentation Sources" at the end of this chapter for the source of each chapter.

Part 1, Using the Network

Chapter 1, "Communications Overview." The rest of this chapter explains why you may want to communicate over a network, and briefly describes the types of communications facilities available under the IRIX operating system such as TCP/IP, UUCP, NFS™, electronic mail.

Chapter 2, "Network Utilities," explains how to use network utilities.

Part 2, Network Programming

Chapter 3, "Network Programming," describes how to use interprocess communication (IPC) facilities.

Chapter 4, "RPC Programming," explains how to use remote procedure calls in network applications.

Chapter 5, "*rpcgen* Compiler," describes how to use the *rpcgen* compiler and describes the RPC language.

Chapter 6, "XDR Programming," presents the library routines a programmer can use to describe arbitrary data structures in a machine-independent fashion.

Chapter 7, "RPC Specification," explains the Remote Procedure Call (RPC) protocol.

Part 3, Network Administration

Chapter 8, "Network Administration," explains how to configure the network communications software.

Chapter 9, "The BIND Name Server," describes how to use the Berkeley Internet Name Domain (BIND) server.

Appendix A, "The Mail System," describes how to set up network mail.

1.4 What Is a Network?

A network is a way to link together a group of computers and other devices (such as printers) so they can share and transfer information. A program running on one computer can interact via the network with its peer on another computer to share data or to control a device.

You may want to use a network for any number of reasons. For example, you can send and receive electronic mail, transfer files from one computer to another (e.g., a report or scientific data generated on a mainframe computer), and access and use files on another computer as if they were on your machine's disks. In addition, you can use the network to access and control devices such as a printer or laboratory device.

1.5 Types of Network Communication

Your computer, such as an IRIS-4D, is able to communicate with another computer through either a network such as an Ethernet or a serial line. This section compares Ethernet and serial-line networks and the software that runs on each.

To communicate across a physical link, two or more computer systems use a communication "protocol," which is simply a procedure that has a well-defined format for transmitting data.

Once connected to an Ethernet or serial-line network, you can use the Internet Protocol (IP) suite, of which the Transmission Control Protocol/Internet Protocol (TCP/IP) is the most important, or the DECnet™ protocol suite with the 4DDN optional product. The optional Network File System (NFS) is based on the IP suite. If you are connected to a serial network, you can also use the UNIX-to-UNIX Copy Program (UUCP).

1.5.1 Ethernet and Serial-Line Networks

To connect your computer to an Ethernet network, you must have this hardware:

- an Ethernet board installed
- a drop cable connected to a transceiver
- an Ethernet cable

The Ethernet board lets you communicate on Ethernet networks. Each computer on the Ethernet network is connected to the Ethernet cable by a drop cable.

Figure 1-1 shows computers that are connected to an Ethernet network.

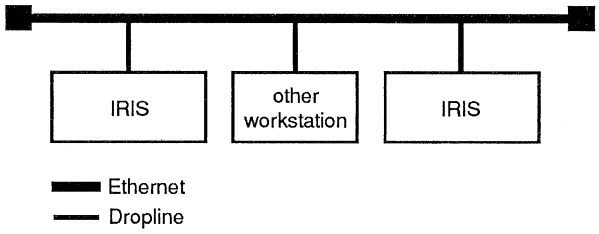


Figure 1-1. Computers Connected by an Ethernet

Serial-line networks allow computers to communicate with other computers that are connected by serial lines and/or modems. A modem is a device that allows computers to transmit data over telephone lines. You do not need special hardware installed in your computer to use serial networks.

Figure 1-2 shows computers that are connected by serial lines.

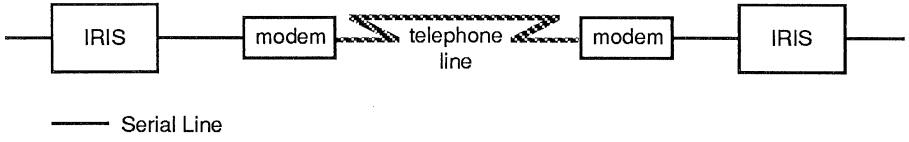


Figure 1-2. Computers Connected Through Serial Lines

The table that follows shows the communication mechanisms that are standard and optional under the IRIX operating system.

Standard Communication Mechanisms	Optional Communication Mechanisms
TCP/IP software	SL/IP software
UUCP software	NFS software
Mail software	4DDN software

Table 1-2. Standard and Optional Communication Mechanisms

1.5.2 The Internet Protocol Suite

The Internet Protocol suite is a collection of layered protocols developed by the U.S. Defense Advanced Research Projects Agency (DARPA). The two most widely used protocols in the IP suite are the Transmission Control Protocol (TCP/IP) and User Datagram Protocol (UDP/IP).

TCP/IP provides a reliable means of transferring data with other systems running TCP/IP on your network. TCP/IP generally provides a higher level of performance than UUCP. UDP/IP provides a faster, low-overhead but unreliable method of transferring data.

By using network applications built on top of the IP suite, you can interactively:

- transfer files between computers
- log in to remote computers and start a shell
- execute commands on remote computers
- send mail between users

The IRIX operating system network layers look like this:

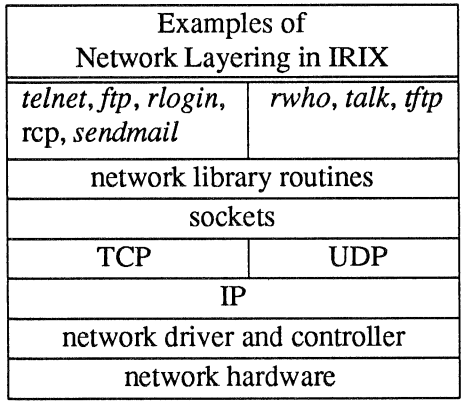


Figure 1-3. IRIX Network Layers

TCP creates a "virtual circuit." A virtual circuit is a data path in which data blocks are guaranteed to be delivered to the target machine in the correct order. Messages are sent from the sender to the receiver until the receiver sends back a message saying that all the data blocks have been received in the correct order.

For details on using Internet network commands, see Chapter 2. For information on Internet programming and configuration, see Parts 2 and 3 in this manual.

1.5.3 UUCP Communication

UNIX-to-UNIX-Copy Program (UUCP) is a UNIX utility that lets your system communicate with remote systems over a serial network.

By using UUCP commands, you can:

- transfer files between local and remote systems in batch mode
- execute programs on remote systems in batch mode
- send mail between local and remote systems in batch mode
- access the USENET network news service

An illustration of a UUCP network appears in Figure 1-4.

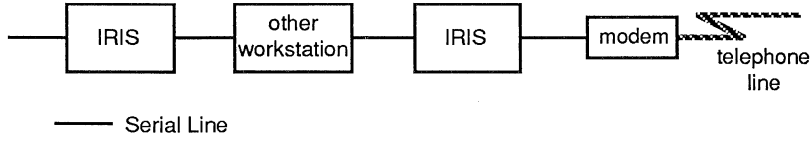


Figure 1-4. UUCP Network

Refer to "Basic Networking" in the *IRIX System Administrator's Guide* or "Communication Tutorial" in the *IRIS-4D User's Guide* for information on network configuration and UUCP commands. See "The Mail System" in Appendix A for information on sending electronic mail with UUCP. For information on using UUCP with modems, see the documents listed previously and the documents that are supplied with your modem.

1.5.4 Choosing TCP/IP or UUCP

Both UUCP and Internet software are standard on the IRIX operating system. To use the Internet software, you must be connected to an Ethernet network or have the Serial-Line IP (SL/IP) optional software and a modem. To use UUCP, you must be connected to a serial network. This section compares UUCP and the Internet suite (e.g., TCP/IP).

TCP/IP provides reliable interactive services.

UUCP is a batch-mode service; when you issue a UUCP command, it is placed in a queue with other commands. The system checks the queue at regular intervals and executes the commands that it finds. After your command is carried out, UUCP reports the results of the command to you. The time it takes to carry out a command on a remote machine varies on different systems.

The table that follows compares features of TCP/IP and UUCP.

TCP/IP Features	UUCP Features
runs on Ethernet network or over serial lines	runs on serial network
transfers files interactively	transfers files in batch mode
executes commands interactively on remote systems	executes commands on remote systems in batch mode
interactively sends mail	sends mail in batch mode
starts a shell on a remote system	starts a shell on a remote system

Table 1-3. Comparison of Features of TCP/IP and UUCP

1.5.5 The Mail System

You can send messages to and receive messages from other users on your system or on your network by using electronic mail.

You can send mail through either UUCP or TCP/IP. Use TCP/IP if your system is connected to an Ethernet network; use UUCP if you are connected to a serial network. Silicon Graphics, Inc., uses System V */bin/mail* and 4.3BSD *sendmail* and */usr/sbin/Mail* for its mail implementation.

For information on using network mail, see Chapter 2; for information on setting up mail, turn to Appendix A.

1.5.6 Network File System

NFS is an optional software package that allows the IRIS to transparently access files on remote computers, including computers from other manufacturers, as if they were on a local disk. This offers two advantages:

- you can share programs developed or maintained on another system
- you can save space on your system

For example, you can store manual pages on only one system and all other systems can access them; you can use them as if they resided on your system.

Figure 1-5 shows a user on a system named *camellia* who requests the manual pages from system named *peony*. NFS makes the manual pages available to the user on *camellia*.

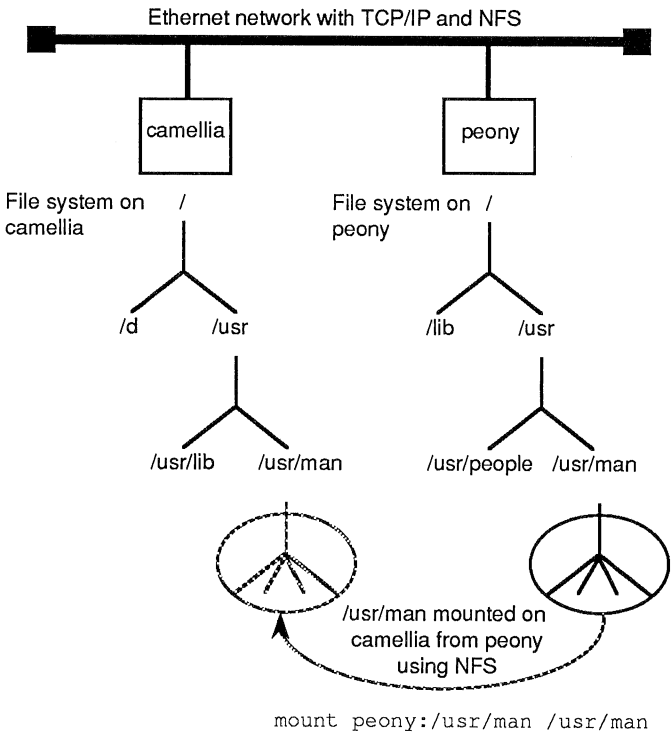


Figure 1-5. Sharing Manual Pages Across the Network Using NFS

For complete information on installing and using NFS, refer to the current *NFS Release Notes* and the *NSF User's Guide.*

1.6 Networking with Non-UNIX Hosts

Many non-UNIX based computers now have implementations of the TCP/IP protocol on their machines. For example, DEC's VAX™ under VMS™, IBM mainframes under VM™ or MVS™, PCs under MS-DOS™, and the Macintosh® implement a subset of the Internet and Berkeley network utilities.

Silicon Graphics, Inc. also offers a number of non-UNIX derived network communication options including 4DDN, SNA, 4D-3270 Emulator, and IRIS-5080 Emulator products. 4DDN is an optional software package that allows the IRIS to communicate with VAXes and other computers using DECnet protocols.

For more information, contact your sales organization.

1.7 Product Support

Silicon Graphics, Inc., provides a comprehensive product support and maintenance program for its products. For further information, contact your service organization.

1.7.1 CMC Ethernet Controller Upgrades

Certain IRIX systems use the CMC ENP-10 Ethernet controller. Early versions of this controller contain firmware that does not support Raw sockets, IP multicasting, or 4DDN. To use these features, you will have to upgrade the firmware.

Use the *hinv*(1) command to verify the version of the controller's firmware. If the *hinv* output shows:

```
CMC ENP-10 Ethernet controller 0, firmware version 0 (CMC)
```

then contact your product support organization and request a firmware upgrade. The information about the upgrade is in *Field Bulletin 87*.

1.8 Documentation Conventions

This guide uses the standard UNIX convention for referring to entries in the IRIX documentation. The entry name is followed by a section number in parentheses. For example, *rcp*(1C) refers to the *rcp* manual entry in Section 1 of the *IRIX User's Reference Manual.*

In command syntax descriptions and examples, you will see the following conventions:

- *italics* represent a variable parameter, which you replace with a value appropriate for the application or a string. In text descriptions, file names and IRIX commands appear in *italics*.
- `typewriter font` shows command syntax descriptions and examples
- square brackets ([]) surrounding an argument indicate an optional argument

1.9 Documentation

This section lists relevant documentation and documentation sources for this guide.

1.9.1 Relevant Documentation

You may find useful information to help you plan and set up your network in these documents:

- *IRIS-4D Series Owner's Guide*
- *Personal IRIS Owner's Guide*
- *Server Owner's Guide*
- *IRIX User's Reference Manual*
- *IRIX System Administrator's Reference Manual*
- *IRIX Programmer's Reference Manual*
- *Mail Reference Manual*
- *NFS User's Guide* and *Administrator's Guide for Diskless Workstations* if the NFS option is installed
- Internet *Request For Comment* documents in the *Defense Data Network Protocol Handbook,* are available from the DDN Network Information Center, SRI International, 333 Ravenswood Avenue, Menlo Park, California 94025, telephone: 1-800-235-3155 or 1-415-859-3695.

1.9.2 Documentation Sources

The documentation sources for each chapter and this guide are given below.

Chapter 1, "Communications Overview," is derived from the *IRIS-4D Series Communications Guide.*

Chapter 2, "Network Utilities," is from the *TCP/IP User's Guide* for the IRIS-4D.

Chapter 3, "Network Programming," is based on chapters 7 and 8 in the *4.3BSD UNIX Programmer's Supplementary Documents, Volume 1*: "An Introductory 4.3BSD Interprocess Communication Tutorial" by Stuart Sechrest, and "An Advanced 4.3BSD Interprocess Communication Tutorial" by Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, Steve Miller and Chris Torek.

Chapter 4, "RPC Programming," is based on Sun Microsystems *Remote Procedure Call Programming Guide* for RPC4.0 and was formerly in the *NFS User's Guide.*

Chapter 5, "*rpcgen* Compiler," is based on Sun Microsystems *rpcgen Programming Guide* for RPC4.0 and was formerly in the *NFS User's Guide.*

Chapter 6, "XDR Programming," is based on Sun Microsystems *External Data Representation: Sun Technical Notes* and *External Data Representation Standard: Protocol Specification* for RPC4.0 and was formerly in the *NFS User's Guide.*

Chapter 7, "RPC Specification," is based on Sun Microsystems *Remote Procedure Calls: Protocol Specification* for RPC4.0 and was formerly in the *NFS User's Guide.*

Chapter 8, "Network Administration," is based on Chapter 1 in the 4.3BSD UNIX *System Manager's Manual* by Michael J. Karels, Chris Torek, James M. Bloom, Marshall Kirk McKusick, Samuel J. Leffler, and William N. Joy.

Chapter 9, "The BIND Name Server," is based on Chapter 11 of the 4.3BSD UNIX *System Manager's Manual* by Kevin J. Dunlap.

Appendix A, "The Mail System," is from the *Communications Guide.*

In addition to the references mentioned above, this guide uses portions of the following documents:

Deering, S. "Host Extensions for IP Multicasting." *Internet Request For Comment 1112.* Network Information Center, SRI International, Menlo Park, California. August 1989.

Lottor, M., "Domain Administrator's Guide." *Internet Request For Comment 1033.* Network Information Center, SRI International, Menlo Park, California. November 1987.

Lottor, M. "TCP Port Service Multiplexer (TCPMUX)." *Internet Request For Comment 1078.* Network Information Center, SRI International, Menlo Park, California. November 1988.

Mockapetris, P., "DNS Encoding of Network Names and Other Types." *Internet Request For Comment 1101.* Network Information Center, SRI International, Menlo Park, California. April 1989.

Mockapetris, P., "Domain Names – Concept and Facilities." *Internet Request For Comment 1034*. Network Information Center, SRI International, Menlo Park, California. November 1987.

Mockapetris, P., "Domain Names – Implementation and Specification." *Internet Request For Comment 1035*. Network Information Center, SRI International, Menlo Park, California. November 1987.

Romano, S., Stahl, M., Recker, M. "Internet Numbers." *Internet Request For Comment 1117*. Network Information Center, SRI International, Menlo Park, California. August 1989.

Stahl, M., "Domain Administrator's Guide." *Internet Request For Comment 1032*. Network Information Center, SRI International, Menlo Park, California. November 1987.

2. Network Utilities

This chapter describes the most frequently used network utilities. You'll want to use these utilities to access another machine on the network.

One set of utilities, the 4.3BSD *r* (remote) commands, work on any system based on 4.2BSD or 4.3BSD, such as the IRIX system. These commands include:

Command:	Use this command to:
rcp	copy a file from one computer running UNIX to another computer running UNIX
rsh	execute a command on a remote host running UNIX
rlogin	initiate a login on a remote host running UNIX
rwho	display a list of the current users on remote UNIX hosts
ruptime	display the status of remote UNIX systems

Table 2-1. The *r* (remote) Network Commands

The commands listed below use standard Internet protocols for virtual terminal support and file and mail transfer. You can use these commands between any systems, UNIX and non-UNIX (and between all intervening networks and gateways), that support these protocols.

Command:	Use this command to:
mail	send electronic mail
telnet	initiate a login on a remote host
ftp	transfer files between hosts

Table 2-2. Other Network Commands

2.1 Controlling Access with *.rhosts*

The 4.3BSD *r* commands use the *.rhosts* file to permit remote access on your system without using passwords. You (or the superuser) create the *.rhosts* file. It belongs in your home directory and specifies the remote systems and users that can access the local system under your login ID. If you are listed in another user's *.rhosts* file on a remote machine, you can use *rcp*, *rsh*, and *rlogin* on that machine with your login ID, and you have the same privileges as the user who listed you in their *.rhosts* file.

Each line of a *.rhosts* file contains the host name of a remote system and a user on that system who can access the local system. Host and user names are separated by spaces and/or tabs, for example:

```
orchid  terry
tulip   kim
```

You or the superuser must own your *.rhosts* file. It must only be writable by the owner (you or the superuser), otherwise, the file is ignored. Add the proper protection using:

```
chmod go-w .rhosts
```

The *.rhosts* file is used to validate a user only when the name of the remote system does not exist in the */etc/hosts.equiv* file. See Chapter 8, "Network Administration," and the *rhosts*(4) and *hosts.equiv*(4) manual pages for more information.

2.2 Using *rcp*, the Remote Copy Program

The *rcp* program copies a file from one system to another.

The following pages explain how to use *rcp* from the command line. In addition to entering commands from the command line, you can also use your WorkSpace to transfer files to and from a remote machine. The WorkSpace has an easy-to-use visual interface for transfering files: the Transfer Manager. For information on the WorkSpace and Transfer Manager, refer to the Owner's Guide for your particular machine, e.g., *IRIS-4D Owner's Guide* or *Personal IRIS Owner's Guide.*

To use the *rcp* program, specify the source machine, user, and the pathname for the file, followed by the destination machine, user, and the destination pathname for the file. The syntax is:

```
rcp  local_pathname  [user@]destination:[remote_pathname]
rcp  [user@]source:remote_pathname  local_pathname
rcp  [user1@]source:pathname  [user2@]destination:[pathname]
```

The square brackets indicate that the information contained within them is optional. When using the *rcp* program:

- if you do not specify a name for the source or destination machine, the system assumes the local machine
- if you do not specify a pathname for the destination machine, the system places the file in the user's home directory on that machine
- if you do not specify a user name with the remote machine name, the system uses your user name

If you want to copy a set of files specified by a wildcarded name (*), you must enclose the file name in single or double quotes:

```
rcp  "guest@astor:/usr/src/*.c"  my_src_dir
```

The examples on the following pages use *rcp*. These examples assume that the user has an account on both hosts. If this isn't the case, specify the user's and host's name as:

```
user@host
```

The user names on each machine must be equivalent (through either */etc/hosts.equiv* or the user's *.rhosts* file) or the accounts on the remote machines must not have passwords. The specified host must be listed in the *hosts*(4) database.

When remotely copying files, you may want to use *rcp* with the –v flag, which copies files in verbose mode, i.e., *rcp* displays a list of the files being transferred.

2.2.1 Copying Files from Local to Remote Machines

The following example copies the local file *sqiral.c* in the current directory to the file *sqiral.c* in the directory */oh4/src/install* on a destination machine named *mint*. The system assumes that the file is on the local machine, because no machine is specified for the file *sqiral.c*.

```
rcp  sqiral.c  mint:/oh4/src/install/sqiral.c
```

If the file exists on the remote host but cannot be written to, *rcp* will print the error message "Permission denied." This means that *rcp* could not replace the file. This message also occurs when you don't have the correct permissions to create the file in the target directory.

2.2.2 Copying Files from Remote to Local Machines

The following example copies the file */usr/include/stdio.h* from the remote machine *violet* to the file *test.h* on the local machine.

```
rcp  violet:/usr/include/stdio.h  test.h
```

2.2.3 Copying Files between Remote Machines

The following example copies the file *user.ps* from your home directory on the remote machine named *rose* to your home directory on the machine named *violet.*

```
rcp  rose:user.ps  violet:
```

2.2.4 Copying Directory Trees

You can use the **–r** option to copy an entire directory tree (a directory and all its subdirectories) between local and remote machines. The following example copies all files and directories from */usr/src/cmd* on the remote machine *violet* to the directory named *newsrc* on the local host.

```
rcp  -r  violet:/usr/src/cmd  newsrc
```

2.3 Using *rsh*, the Remote Shell Program

The *rsh* program connects your system to a remote host and executes the commands you specify on the remote host. Like *rcp*, this network utility assumes that you have accounts with the same user name on both the remote and local hosts or are listed in the remote user's *.rhosts* file. This is the *rsh* syntax:

```
rsh  hostname  [-l  username]  command
```

For example, to find who is logged in on another machine, type:

```
rsh  hostname  who
```

If your account name on the remote host is *pat*, type:

```
rsh  hostname  -l  pat  who
```

The specified remote system must be listed in the *hosts*(4) database. If you do not specify a command to execute, *rsh* initiates an *rlogin* (remote login) on the remote machine. (See the section below on using *rlogin*.) Interactive commands such as *vi*(1) do not run correctly when you execute them using *rsh*; use *rlogin* instead.

2.4 Using *rlogin*, the Remote Login Program

The *rlogin* program initiates a login on a remote host across the network. The program takes the remote system name as an argument. This is the *rlogin* syntax:

```
rlogin  hostname  [-l  username]
```

For example, to log in remotely to a system named *mint*, type:

```
rlogin  mint
```

The specified remote system must be listed in the *hosts*(4) database.

In this example, the user logs in to the system *mint* as the user named *rick*. The user is listed in rick's *.rhosts* file on *mint* or *rick*'s account does not have a password.

```
rlogin  mint  -l  rick
```

2.5 Using *rwho*

The *rwho*(1C) command generates a list of users currently logged in on all UNIX systems on the local network. In addition to the user's login name, *rwho* lists the terminal name, the system name, and the login time for each user.

Note: You can only use this command if the *rwhod* daemon is running on the remote UNIX host.

The *rwhod* daemon should not be run on networks with a large number of machines. It can consume significant amounts of system resources when used in large networks. See Chapter 8, "Network Administration," for information on enabling *rwhod.*

To display a list of all users logged in to UNIX hosts on the network, type:

```
rwho
```

This command generates a list of users similar to the following example:

```
percy     mint:tty15      Mar 14 08:17
arnold    mint:ttyq4      Mar 14 08:35
root      knot:console    Mar 12 15:42
helen     troy:tty03      Mar 14 09:38
```

In this display, the first column lists the name of the user; the second column lists the name of the host and the name of the terminal on that host; the third column lists the login date and time.

2.6 Using *ruptime*

The *ruptime* utility displays the status of all UNIX hosts on the local network. Along with the name of the host, *ruptime* displays the current status, how long the system has been up, and the average number of jobs in the run queue for each system. Like *rwho*, *ruptime* depends on the *rwhod* daemon; if the daemon is not running, ruptime is not usable. See Chapter 8, "Network Administration," for information on enabling *rwhod.*

To display the status information for all UNIX systems on the network, type:

```
ruptime
```

The status list displayed is similar to this example:

```
mint     up   12+18:54     4 users, load 1.10, 0.83, 0.75
lily     down 13:25
violet   up   5+02:45     10 users, load 2.51, 1.87, 0.51
```

In the status list, the first column displays the system name, and the second column indicates whether the system is up or down and the amount of time it has been in this condition. The second column uses this format:

```
days + hours:minutes
```

The third column lists the number of users logged in to the system and the average number of jobs in the event queue in the last 5, 10, and 15 minutes.

The *ruptime* program does not report users who are idle for more than one hour unless you use the **–a** flag.

2.7 Using *mail*

The network software extends the IRIX system mail facility to allow users to send mail to other users on the network. To send mail to another user on another machine, first type the user's login name, followed by an at sign (@) and the machine name:

```
mail user_name@machine_name
```

After you finish typing the message, to send it, type <ctrl-d> or a period (.) at the beginning of a new line. For information on enabling network mail, see Chapter 8, "Network Administration." For details on setting up and maintaining the mail system, see Appendix A.

2.8 Using *telnet*, a Remote Login Program

The *telnet* program initiates a virtual terminal session on a remote host across the network. This program uses the Internet TELNET protocol, which is implemented on many UNIX and non-UNIX systems.

To use the *telnet* program, follow these steps:

1. To start the *telnet* program, use the name or Internet address of the remote machine as an argument. For example, to log in remotely to a machine named *rose*, type:

   ```
   telnet rose
   ```

   The screen shows:

   ```
   Trying...,
   Connected to rose.
   Escape character is ^].

   login:
   ```

2. Execute commands on the remote machine.

3. To disconnect from the remote machine, type:

   ```
   logout
   ```

   If you do not get the local machine prompt, exit the *telnet* program by pressing **<ctrl-]>**. This gets the attention of the *telnet* program. At the `telnet>` prompt, type:

   ```
   quit
   ```

The *telnet* program has many options and internal commands. See *telnet*(1C) for details.

2.9 Using *ftp*, the File Transfer Program

The *ftp* program transfers files using the Internet File Transfer Protocol. This section describes some of the major *ftp* commands. For more information, see *ftp*(1C).

The following sequence explains how to enter and exit *ftp*.

1. To start the *ftp* program, type:

   ```
   ftp hostname
   ```

   hostname is the name or Internet address of the machine with which you wish to communicate. For example, to connect to the host named *quake*, type:

   ```
   ftp quake
   ```

   After you execute the above command, the screen shows a message similar to:

   ```
   Connected to quake.
   220 quake FTP server (IRIX version date time) ready.
   ```

2. Next, the host requests your name and password. For example, a user named peter would type:

   ```
   Name (quake:peter): peter
   331 Password required for peter.
   ```

   Note that the remote user account must have a password; *ftp* will not let you log in if the password is missing. Once you enter the proper password, the host replies:

   ```
   230 User peter logged in.
   ```

   After you have logged in, more messages may appear followed by the *ftp* prompt.

   ```
   ftp>
   ```

   To access an anonymous ftp account, when the host requests the name and password, type `anonymous` for the name.

   ```
   Name (quake:peter): anonymous
   331 Guest login ok, send ident as password.
   ```

   It's customary to enter your name and host as the password, for example:

   ```
   peter@quake
   ```

As with regular *ftp* accounts, after you have logged in, more messages may appear followed by the *ftp* prompt.

3. Now you are ready to send and receive files. To see a list of the commands for *ftp*, type:

```
help
```

4. To exit from *ftp* after you finish transferring files, type:

```
quit
```

The sections that follow describe some of the commands that you can use with *ftp*, including both the command syntax and an example of the command. Other commands are available; see *ftp*(1C) for more information.

2.9.1 Transferring Files from Local to Remote Machines

To send one file to the remote host, use this syntax:

```
put  localfile  [remotefile]
```

For example, this command sends the file *myfile* to the remote machine using the same file name as the file on the local machine:

```
put  myfile
```

ftp displays messages to indicate that the file has been sent to the remote machine.

An *ftp* message generated by the above command looks like this:

```
local:  myfile  remote: myfile
200  PORT  command  okay.
150  Opening BINARY mode data connection for 'myfile'
     (2479 bytes).
226  Transfer  complete.
2479 bytes received in 0.06 seconds (40.34 Kbytes/s)
```

To append a file to another file on the remote machine, use this syntax:

```
append  localfile [remotefile]
```

If the remote filename is not given, *ftp* uses the local filename. For example, to append the file *dictionary.a* to *dictionary.b*, type:

```
append  dictionary.a  dictionary.b
```

The current working directory on the remote machine is the assumed destination for the file. If you do not specify a remote file name, the local file name is used as the file name on the remote machine.

To transfer multiple files from the local machine to a directory on the remote machine, type:

```
cd  remote_directory
mput  localfile1 localfile2 ...
```

For example, to transfer the files *thisfile* and *thatfile* to the directory */usr/people/helen/filexfer*, type:

```
cd  /usr/people/helen/filexfer
mput  thisfile  thatfile
```

ftp displays messages to indicate that the files have been sent to the remote machine.

2.9.2 Transferring Files from Remote to Local Machines

To bring a file from the remote machine to the local machine, type:

```
get  remotefile localfile
```

For example, this command gets the file *yourfile* from the remote host and stores it on the local machine in the current directory as *myfile*:

```
get  yourfile  myfile
```

ftp prints messages to indicate that the file has been received by the local machine.

To transfer multiple files from the remote machine to the local machine into the current directory, type:

```
mget  remotefile1 remotefile2 ...
```

For example, to transfer the files *localinclude* and *graph.c*, type:

```
mget  localinclude  graph.c
```

2.9.3 Executing Local Commands

To invoke commands on the local machine, precede the command with an exclamation point (!). For example, to list the files retrieved with the *get* or *mget* commands, type:

```
!ls
```

This command displays a listing of the current directory on the local machine.

To put a listing of remote files in a local file, type:

```
mls  remotefiles localfile
```

For example, to make a listing of the files *local.h*, *float.h*, and *graphics.h* in the file *includefiles* on the local machine, type:

```
mls  local.h  float.h  graphics.h  includefiles
```

To change directories on the local machine, type:

```
lcd  directory
```

For example, to change your local directory to *fish* on the local machine, type:

```
lcd  fish
```

2.9.4 Executing Remote Commands

Commands on the remote machine do not need to be preceded by an exclamation point. The commands explained below fall into this category. To change your working directory on the remote machine, use the command:

```
cd  remote_directory
```

For example, to change your working directory to *localinclude*, type:

```
cd  localinclude
```

To print the current working directory on the remote machine, type:

```
pwd
```

To print the contents of a remote directory, type:

```
dir  [remote_directory]  [localfile]
```

or

```
ls  [remote_directory]  [localfile]
```

If you specify a local file name, the contents of the directory are placed in this local file. If you do not specify a local file name, the list is displayed on the screen. If you do not specify a remote directory, the current working directory is assumed.

2.9.5 Transferring Binary or ASCII Files

You can transfer either binary or ASCII files with *ftp*. The default type depends on the remote system type. To transfer binary files, type:

```
binary
```

After issuing this command, any subsequent files are transferred in binary mode. To return to the default of transferring ASCII files, type:

```
ascii
```

The *ftp* commands have identical syntax for either ASCII or binary file transfer.

2.9.6 Summary of *ftp* Commands

The table that follows lists and briefly describes the *ftp* commands discussed in this section.

To do this:	Use the following syntax:
start *ftp*	ftp *hostname*
list *ftp* commands	help
quit *ftp*	quit
execute local command	!*command*
execute remote command	*command*
transfer binary file	binary
transfer ASCII file	ascii
FROM LOCAL TO REMOTE MACHINES	
transfer a file	put *localfile* [*remotefile*]
append a file	append *localfile remotefile*
transfer multiple files	cd *remote_dir* mput *localfile1 localfile2 ...*
execute command	!*command*
change directory	lcd
FROM REMOTE TO LOCAL MACHINES	
transfer file	get *remotefile localfile*
transfer multiple files	mget *remotefile1 remotefile2 ...*

Table 2-3. *ftp* Commands

3. Network Programming

This chapter describes the 4.3BSD interprocess communication (IPC) facilities in IRIX. It is designed to complement the manual pages for the IPC primitives by providing examples of their use.

This chapter covers the following topics:

- Internet and UNIX domain sockets
- IPC-related system calls and the basic model of communication
- the supporting library routines used to construct distributed applications
- the client/server model used in developing applications and includes examples of the two major types of servers
- advanced topics sophisticated users are likely to encounter when using the IPC facilities, such as IP/UDP broadcasting and multicasting.

3.1 Compiling 4.3BSD Programs

Most 4.3BSD programs compile and link under IRIX without change. Some may require compiler directives, linking with additional libraries or even source code modification.

If the program assumes the `char` data type is signed, use the `-signed` compiler directive. For example:

```
cc -signed example.c -o example -lc_s
```

Most BSD programs assume signed characters. Linking with the shared C library, `-lc_s`, saves space and improves portability.

The BSD library routines formerly in */usr/lib/libbsd.a* are now in the standard C library. Most BSD programs do not need to link with this library — see the (3B) section in *intro*(3) for details.

Several network library routines have Yellow Pages equivalents. Link with `-lsun` to use the Yellow Pages versions if the NFS optional software is installed:

```
cc -signed example.c -o example -lsun -lc_s
```

IRIX provides System V, 4.3BSD and POSIX signal handling mechanisms. BSD signals are obtained with the `-D_BSD_SIGNALS` compiler directive or by modifying the source code to place

```
#ifdef sgi
#define _BSD_SIGNALS
#endif
```

before including *<signal.h>*.

Note: In previous versions of IRIX, BSD header files were located in the directory /usr/include/bsd. Programs that included headers files with the statement:

```
#include <bsd/file.h>
```

were not portable and should be changed to remove "`bsd/`".

3.2 Basics

The *socket* is the basic building block for communication. A socket is an endpoint of communication to which a name can be *bound*. Each socket in use has a *type* and one or more associated processes. The processes communicate through the socket.

Sockets exist within *communication domains*. A domain dictates various properties of the socket. One such property is the scheme used to name sockets. For example, in the UNIX communication domain, sockets are named with UNIX path names; e.g., a socket may be named "/dev/foo."

Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The IRIX socket facilities support three separate communication domains:

- the UNIX domain, which is used for on-system communication
- the Internet domain, which is used by processes that communicate using the Internet standard communication protocols IP/TCP/UDP
- the Raw domain, which provides access to the link-level protocols of network interfaces.

The underlying communication facilities provided by these domains have a significant influence on the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the error conditions that are possible when operating in the Internet domain.

A *protocol* is a set of rules, data formats and conventions that regulate the transfer of data between participants in the communication. In general, there is one protocol for each socket type (stream, datagram, etc.) within each domain. The code that implements a protocol keeps track of the names that are bound to sockets, sets up connections, and transfers data between sockets, perhaps sending the data across a network. It is possible for several protocols, differing only in low level details, to implement the same style of communication within a particular domain. Although it is possible to select which protocol should be used, for nearly all uses it is sufficient to request the default protocol.

3.2.1 Socket Types

Sockets are typed according to visible communication properties. Processes are presumed to communicate only between sockets of the same type, although nothing prevents communication between sockets of different types should the underlying communication protocols support this.

Three types of sockets are available:

- a stream socket
- a datagram socket
- a raw socket

Stream Sockets

A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow and some additional signaling facilities, a pair of connected stream sockets provides an interface similar to that of pipes. (In the UNIX domain, in fact, the semantics are identical.)

Datagram Sockets

A *datagram* socket supports bidirectional flow of messages that are not necessarily sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket can find messages duplicated or in an order different from the order in which they were sent. The data in any single message is in the correct order, with no duplications, deletions, or changes.

An important characteristic of a datagram socket is that record boundaries in the data are preserved. Datagram sockets closely model facilities found in many packet-switched networks. However, datagram sockets provide additional facilities, including routing and fragmentation.

Routing is used to forward messages from one local network to another nearby or distant network. Dividing one large network into several smaller ones can improve network performance in each smaller network, improve security, and facilitate administration and troubleshooting.

Fragmentation divides large messages into pieces small enough to fit on the local medium. It allows application programs to use a single message size independent of the packet size limitations of the underlying networks.

Raw Sockets

A *raw* socket gives you access to the underlying communication protocols that support socket abstractions. Raw sockets are normally datagram-oriented, though their exact characteristics depend on the interface provided by the protocol. Raw sockets are not intended for the general user. They are provided mainly for programmers interested in developing new communication protocols or gaining access to some of the more esoteric facilities of an existing protocol.

3.2.2 Creating Sockets

To create a socket, use the *socket* system call:

```
#include <sys/types.h>
#include <sys/socket.h>
s = socket(domain, type, protocol);
```

This call creates a socket in the specified *domain* and of the specified *type*. You can also request a particular protocol. If the protocol is left unspecified (a value of 0), the system selects an appropriate protocol. It selects from those protocols that comprise the communication domain and that can be used to support the requested socket type. The call returns a descriptor (a small integer) that can be used in later system calls operating on sockets. The domain is specified as one of the manifest constants defined in the file *<sys/socket.h>*. For the UNIX domain the constant is AF_UNIX; for the Internet domain, the constant is AF_INET. For the Raw link-level domain, the constant is AF_RAW. (AF indicates the "address format" to use in interpreting names.) The socket types are also defined in this file as either SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. For example, to create a stream socket in the Internet domain, the following call might be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call creates a stream socket with the default protocol, TCP, providing the underlying communication support. To create a datagram socket for on-machine use, the call might be:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

The default protocol (used when the *protocol* argument to the *socket* call is 0) should be correct for most situations. However, you can specify a protocol other than the default. (See Section 3.5 for details.)

ne 7 To create a "drain" socket that receives all packets having a network-layer type code or encapsulation not implemented by the kernel, the call is:

```
#include <net/raw.h>

s = socket(AF_RAW, SOCK_RAW, RAWPROTO_DRAIN);
```

Raw domain sockets are discussed in detail in the *raw*(7P), *snoop*(7P), and *drain*(7P) manual pages.

The *socket* call can fail for several reasons. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request can fail in response to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

3.2.3 Binding Local Names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages can be received on it. Communicating processes are bound by an *association*. An association is a temporary or permanent specification of a pair of communicating sockets.

In the Internet domain, an association is composed of local and foreign addresses, and local and foreign ports. The structure of Internet domain addresses is defined in the file *<netinet/in.h>*. Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed. When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple address including a protocol as well as the port and machine address.

An association is identified by the tuple <protocol, local address, local port, remote address, remote port>. Duplicate tuples are not allowed. An association may be transient when using datagram sockets; the association actually exists during a *send* operation.

In the UNIX domain, an association is composed of local and foreign pathnames (the phrase "foreign pathname" means a pathname created by a foreign process, not a pathname on a foreign system). UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate <protocol, local pathname, foreign pathname> tuples. The pathnames may not refer to files already existing on the system. Like pathnames for normal files, they may be either absolute (e.g., /dev/imaginary) or relative (e.g., socket). Because these names are used to allow processes to rendezvous, relative pathnames can pose difficulties and should be used with care.

When a name is bound into the name space, a file (inode) is allocated in the file system. If the inode is not deallocated, the name will continue to exist even after the bound socket is closed. This can cause subsequent runs of a program to find that a name is unavailable, and can cause directories to fill up with these objects. The names are removed by calling *unlink*(2) or using the *rm*(1) command. Names in the UNIX domain are only used for rendezvous. They are not used for message delivery once a connection is established. Therefore, in contrast with the Internet domain, unbound sockets need not be (and are not) automatically given addresses when they are connected.

The *bind* system call allows a process to specify half of an association, <local address, local port> (or <local pathname>), while the *connect* and *accept* primitives are used to complete a socket's association.

The *bind* system call is used as follows:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string that is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties that comprise the "domain."

In the UNIX domain, names contain a pathname and a family, which is always AF_UNIX. The following code fragment binds the name "/tmp/foo" to a UNIX domain socket.

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr, strlen(addr.sun_path) +
    sizeof (addr.sun_family));
```

Note that in determining the size of a UNIX domain address, null bytes are not counted, which is why *strlen* is used. In the current implementation of UNIX domain IPC under IRIX, the file name referred to in *addr.sun_path* is created as a socket in the system's file space. The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and this file should be deleted by the caller when it is no longer needed using the *unlink*(2) system call. Future versions of IRIX may not create this file.

In the Internet domain, binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the *connect* and *send* calls automatically bind an appropriate address if they are used with an unbound socket.

In binding an Internet address, use the *bind* system call:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The selection of what to place in the address *sin* requires some discussion. Section 3.3 describes formulating Internet addresses and discusses the library routines used in name resolution.

3.2.4 Establishing Connections

Connection establishment is usually asymmetric, with one process a "client" and the other a "server." The server, when it offers its advertised services, binds a socket to a well-known address associated with the service and then passively "listens" on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a "connection" to the server's socket. On the client side the *connect* call is used to initiate a connection.

Using the UNIX domain, this might appear as:

```
struct sockaddr_un server;
 ...
connect(s, (struct sockaddr *)&server,
        strlen(server.sun_path) +
        sizeof (server.sun_family));
```

Using the Internet domain, this might appear as:

```
struct sockaddr_in server;
 ...
connect(s, (struct sockaddr *)&server, sizeof (server));
```

In the example above, *server* would contain either the UNIX pathname or the Internet address and port number of the server to contact. If the client process's socket is unbound at the time of the *connect* call, the system will automatically select and bind a name to the socket if necessary. This is the usual way that local addresses are bound to a socket.

The *connect* call returns an error if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer can begin. Some of the more common errors returned when a connection attempt fails are listed below:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system stopped trying. This usually occurs because the destination host is down, or because problems in the network resulted in lost transmissions.

ECONNREFUSED
: The host refused service. This is usually because a server process is not present at the requested port on the host.

ENETDOWN or EHOSTDOWN
: These operational errors describe status information delivered to the client host by the underlying communication services.

ENETUNREACH or EHOSTUNREACH

These operational errors can occur either because the network or host is unknown (no route to the network or host is present), or because of status information returned by intermediate gateways or switching nodes. Many times the status returned is not sufficient to distinguish a network being down from a host being down, in which case the system indicates the entire network is unreachable.

For the server to receive a client's connection, it must perform two steps after binding its socket. The first is to indicate that it is ready to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter of the *listen* call specifies the maximum number of outstanding connections that can be queued awaiting acceptance by the server process; this number can be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but the individual messages that comprise the request will be ignored. This gives a busy server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not.

It is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the *listen* call is currently limited by the system to a maximum of 5 pending connections on any one queue. This limit avoids the problem of processes monopolizing system resources by setting an infinite backlog, then ignoring all connection requests.

With a socket marked as listening, a server can *accept* a connection:

```
struct sockaddr_in from;
int fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

(For the UNIX domain, *from* would be declared as a *struct sockaddr_un*, but nothing different would need to be done as far as *fromlen* is concerned. The examples that follow describe only Internet routines.) A new descriptor is returned on receipt of a connection (along with a new socket). To find out who its client is, a server can supply a buffer for the client socket's name. The server initializes the value-result parameter *fromlen* to indicate how much space is associated with *from*. The parameter is then modified on

return to reflect the true size of the name. If the client's name is not of interest, the second parameter can be a null pointer.

Accept normally blocks. That is, *accept* will not return until a connection is available or the system call is interrupted by a signal to the process. Furthermore, a process cannot indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered in Section 3.4.

3.2.5 Transfering Data

IRIX has several system calls for reading and writing information. The simplest calls are *read*(2) and *write*(2). They take as arguments a descriptor, a pointer to a buffer containing the data and the size of the data:

```
char buf [100];
  ...
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

The descriptor may indicate either a file or a connected socket. "Connected" can mean either a connected stream socket or a datagram socket for which a *connect* call has provided a default destination (described below). *Write* requires a connected socket since no destination is specified in the parameters of the system call. *Read* can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that require no assumptions about the source of their input or the destination of their output. There are variations on *read* and *write* that allow the source and destination of the input and output to use several separate buffers, while retaining the flexibility to handle both files and sockets. These are *readv*(3) and *writev*(3), for read and write *vector*.

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. For example, a user interface process may be interpreting commands and sending them on to another process through a stream connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to

send it as *out-of-band* (OOB) data, which is discussed in the "Advanced Topics" section. There are a pair of calls similar to *read* and *write* that allow options, including sending and receiving OOB information; these are *send*(2) and *recv(2)*.

```
send(s, buf, sizeof (buf), flags);
recv(s, buf, sizeof (buf), flags);
```

These calls are used only with sockets; specifying a descriptor for a file will result in the return of an error status. While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important.

The flags, defined in *<sys/socket.h>*, can be a non-zero value if one or more of the following is required:

MSG_PEEK	look at data without reading
MSG_OOB	send/receive out-of-band data
MSG_DONTROUTE	send data without routing packets

To preview data, specify MSG_PEEK with a *recv* call. It allows a process to read data without removing the data from the stream. That is, the next *read* or *recv* call applied to the socket will return the data previously previewed. One use of this facility is to read ahead in a stream to determine the size of the next item to be read. Out-of-band data are specific to stream sockets, and are discussed in the "Advanced Topics" section of this chapter. The option to have data sent in outgoing packets without routing is used only by the routing table management process.

To send datagrams, one must be allowed to specify the destination. The call *sendto(2)* takes a destination address as an argument and is therefore used for sending datagrams. The call *recvfrom*(2) is often used to read datagrams, since this call returns the address of the sender, if it is available, along with the data. If the identity of the sender does not matter, one may use *read* or *recv*.

Finally, there are a pair of calls that allow the sending and receiving of messages from multiple buffers, when the address of the recipient must be specified. These are *sendmsg*(2) and *recvmsg*(2). These calls are actually quite general and have other uses, including, in the UNIX domain, the transmission of a file descriptor from one process to another.

3.2.6 Discarding Sockets

A socket can be discarded by closing the descriptor:

```
close(s);
```

If data are associated with a socket that promises reliable delivery (e.g., a stream socket) when a close takes place, the system will continue trying to transfer the data. However, after a period of time, undelivered data are discarded. Should you have no use for any pending data, perform a *shutdown* on the socket prior to closing it:

```
shutdown(s, how);
```

The value *how* is 0 if you do not want to read data, 1 if no more data will be sent, or 2 if no data are to be sent or received.

3.2.7 Connectionless Sockets

The sockets described so far follow a connection-oriented model. However, connectionless interactions, typical of the datagram facilities found in contemporary packet-switched networks, are also supported. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as described earlier in this chapter. If a particular local address is needed, the *bind* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data are first sent.

To send data, use the *sendto* primitive:

```
sendto(s, buf, buflen, flags,
            (struct sockaddr *)&to, sizeof(to));
```

The *s*, *buf*, *buflen*, and *flags* parameters are used as described earlier for the *send* call. The *to* value indicates the destination address. On an unreliable datagram interface, errors probably will not be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return −1 and the global value *errno* will contain an error number.

To receive messages on an unconnected datagram socket, use the *recvfrom* primitive:

```
recvfrom(s, buf, buflen, flags,
            (struct sockaddr *)&from, &fromlen);
```

Once again, the value-result parameter, *fromlen*, initially contains the size of the *from* buffer and is modified on return to indicate the actual size of the address from which the datagram was received.

In addition to the two calls mentioned above, datagram sockets can also use the *connect* call to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. Only one connected address is permitted for each socket at one time: A second *connect* will change the destination address, and a *connect* to a null address (family AF_UNSPEC) will cause a disconnection. Connect requests on datagram sockets return immediately, as this simply results in the system recording the peer's address (as compared to a stream socket, where a connect request initiates establishment of an end-to-end connection). The *accept* and *listen* calls are not used with datagram sockets.

While a datagram socket is connected, errors from recent *send* calls can be returned asynchronously. These errors can be reported on subsequent operations on the socket, or a special socket option used with *getsockopt*, SO_ERROR, can be used to interrogate the error status. A *select* for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other details of datagram sockets are described in Section 3.5.

3.2.8 Input/Output Multiplexing

You can multiplex I/O requests among multiple sockets and/or files by using the *select* call:

```
#include <sys/time.h>
#include <sys/types.h>
 ...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

The *select* call takes as arguments pointers to three sets: one for the set of file descriptors the caller wants to read data on, one for descriptors data are to be written on, and one for which exceptional conditions are pending. Out-of-band data are the only exceptional condition currently implemented by the socket. If you are not interested in certain conditions (i.e., read, write, or exceptions), the corresponding argument to the *select* call should be a null pointer.

Each set is a structure containing an array of long integer bit masks. The size of the array is set by the definition FD_SETSIZE. The array must be long enough to hold one bit for each of FD_SETSIZE file descriptors.

Use the macros FD_SET(*fd, &mask*) and FD_CLR(*fd, &mask*) to add and remove file descriptor *fd* in the set *mask*. The set should be zeroed before use. To clear the set *mask*, use the macro FD_ZERO(*&mask*). The parameter *nfds* in the *select* call specifies the range of file descriptors (one plus the value of the largest descriptor) to be examined in a set.

You can specify a timeout value if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a poll, returning immediately. If *timeout* is a null pointer, the selection will block indefinitely. To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

The *select* call normally returns the number of file descriptors selected. If the *select* call returns because the timeout expires, the value 0 is returned. If the *select* call terminates because of an error or interruption, a –1 is returned with the error number in *errno*, and with the file descriptor masks unchanged.

For a successful return, the three sets will indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending. The status of a file descriptor in a select mask can be tested with the FD_ISSET(*fd, &mask*) macro. This macro returns a non-zero value if *fd* is a member of the set *mask*, and 0 if it is not.

To check for read readiness on a socket to be used with an *accept* call, use *select* followed by an FD_ISSET(*fd, &mask*) macro. If FD_ISSET returns a non-zero value, indicating permission to read, then a connection is pending on the socket.

For example, to read data from two sockets, *s1* and *s2*, as the data are available and with a one-second timeout, the following code might be used:

```
#include <sys/time.h>
#include <sys/types.h>
 ...
fd_set read_template;
struct timeval wait;
 ...
for (;;) {
    wait.tv_sec = 1;                /* one second */
    wait.tv_usec = 0;

    FD_ZERO(&read_template);

    FD_SET(s1, &read_template);
    FD_SET(s2, &read_template);

    nb = select(FD_SETSIZE, &read_template,
            (fd_set *) 0, (fd_set *) 0, &wait);

    if (nb <= 0) {
        /*
         * An error occurred during the select, or
         * the select timed out.
         */
    }

    if (FD_ISSET(s1, &read_template)) {
        /* Socket #1 is ready to be read from. */
    }

    if (FD_ISSET(s2, &read_template)) {
        /* Socket #2 is ready to be read from. */
    }
}
```

In 4.2BSD, the arguments to *select* were pointers to integers instead of pointers to *fd_sets*. This type of call will still work as long as the largest file descriptor is itself numerically less than the number of bits in an integer (i.e., 32). However, the methods illustrated above should be used in all current programs.

The *select* call provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals.

3.3 Network Library Routines

Programs need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. This section discusses the routines provided to manipulate Internet network addresses.

Locating a service on a remote host requires many levels of mapping before client and server can communicate. A service is assigned a name, such as "*login server*," that people can easily understand. This name, and the name of the peer host, must then be translated into network *addresses*. Finally, the address is used in locating a physical *location* and *route* to the service. The specifics of these three mappings can vary between network architectures. For instance, it is desirable for a network to not require hosts to have names indicating their physical location to the client host. Instead, underlying services in the network can discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location-independent manner can induce overhead in connection establishment, as a discovery process must take place, but it allows a host to be physically mobile. The host does not have to notify its clients of its current location.

Standard routines are provided for mapping:

- host names to network addresses
- network names to network numbers
- protocol names to protocol numbers
- service names to port numbers

Routines also indicate the appropriate protocol to use in communicating with the server process. The file *<netdb.h>* must be included when using any of these routines.

3.3.1 Host Names

The *hostent* data structure provides an Internet host name-to-address mapping:

```
struct   hostent {
    char  *h_name;        /* official name of host */
    char  **h_aliases;    /* alias list */
    int   h_addrtype;     /* host address type (e.g., AF_INET) */
    int   h_length;       /* length of address */
    char  **h_addr_list; /* list of addresses, null terminated */
};
/* first address, network byte order */
#define h_addr    h_addr_list[0]
```

The routine *gethostbyname*(3N) takes an Internet host name and returns a *hostent* structure, while the routine *gethostbyaddr*(3N) maps Internet host addresses into a *hostent* structure.

These routines return the official name of the host and its public aliases, along with the address type (family) and a null-terminated list of variable length address. This list of addresses is required because it is possible for a host to have many addresses and the same name. The *h_addr* definition is provided for backward compatibility, and is defined to be the first address in the list of addresses in the *hostent* structure.

The database for these calls is provided either by the file */etc/hosts* (see *hosts*(4)), or by use of the Internet domain name server, *named*(1M). The database can also come from the Yellow Pages, if you have the NFS option. Because of the differences in these databases and their access protocols, the information returned can differ. When using the host table YP versions of *gethostbyname*, the call returns only one address but includes all listed aliases. When using the name server version, the calls can return alternate addresses, but will not provide any aliases other than one given as argument.

3.3.2 Network Names

Routines for mapping network host names to numbers, and back, are provided. These routines return a *netent* structure:

```
/*
 * Assumption here is that a network number
 * fits in 32 bits.
 */
struct netent {
    char  *n_name;      /* official name of net */
    char  **n_aliases;  /* alias list */
    int   n_addrtype;   /* net address type */
    int   n_net;        /* network number, host byte order */
};
```

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above. The routines extract their information from */etc/networks* or from the Yellow Pages, if the NFS option is installed.

3.3.3 Protocol Names

The *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

```
struct   protoent {
    char  *p_name;      /* official protocol name */
    char  **p_aliases;  /* alias list */
    int   p_proto;      /* protocol number */
};
```

The routines extract their information from */etc/protocols* or from the Yellow Pages if the NFS option is installed.

3.3.4 Service Names

A service is expected to reside at a specific "port" and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Furthermore, a service can reside on multiple ports. If it does, the higher-level library routines will have to be bypassed or extended.

Services available are obtained from the file */etc/services* or from the Yellow Pages if the NFS option is installed.

The *servent* structure describes a service mapping:

```
struct   servent {
    char  *s_name;      /* official service name */
    char  **s_aliases; /* alias list */
    int   s_port;       /* port #, network byte order */
    char  *s_proto;     /* protocol to use */
};
```

The routine *getservbyname*(3N) maps service names to a servent structure by specifying a service name and, optionally, a qualifying protocol. Thus:

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification for a telnet server using any protocol, while:

```
sp = getservbyname("telnet", "tcp");
```

returns only the telnet server that uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) also provide service mappings. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; specify an optional protocol name to qualify lookups.

3.3.5 Network Dependencies

With the support routines described above, an Internet application program rarely has to deal directly with addresses. This allows services to operate as much as possible in a network-independent fashion. However, purging all network dependencies is difficult. As long as the user must supply network addresses when naming services and sockets, some network dependency is required in a program. For example, the normal code included in client programs, e.g., the remote login program, takes the form shown in Figure 3-1.

To make the remote login program independent of the Internet protocols and addressing scheme, a program would have to have a layer of routines that masks the network-dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear worthwhile.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char *argv[];
{
    struct sockaddr_in server;
    struct servent *sp;
    struct hostent *hp;
    int s;
    ...
    sp = getservbyname("login", "tcp");
    if (sp == NULL) {
        fprintf(stderr,
            "rlogin: tcp/login: unknown service\n");
        exit(1);
    }
    hp = gethostbyname(argv[1]);
    if (hp == NULL) {
        fprintf(stderr,
            "rlogin: %s: unknown host\n", argv[1]);
        exit(2);
    }
    bzero((char *)&server, sizeof (server));
    bcopy(hp->h_addr, (char *)&server.sin_addr,
            hp->h_length);
    server.sin_family = hp->h_addrtype;
    server.sin_port = sp->s_port;
    s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) {
        perror("rlogin: socket");
        exit(3);
    }
    ...

    /* Connect does the bind() for us */
    if (connect(s, (char *)&server,
                sizeof (server)) < 0) {
        perror("rlogin: connect");
        exit(4);
    }
    ...
}
```

Figure 3-1. Remote Login Client Code

3.3.6 Byte Ordering

Aside from the address-related data-base routines, several other routines are available to simplify manipulation of names and addresses. Table 3-1 summarizes the routines for manipulating variable-length byte strings and handling byte swapping of network addresses and values.

Call	Synopsis
bcmp(s1, s2, n)	compare byte-strings; 0 if same, not 0 otherwise
bcopy(s1, s2, n)	copy n bytes from s1 to s2
bzero(base, n)	zero-fill n bytes starting at base
htonl(val)	convert 32-bit quantity from host to network byte order
htons(val)	convert 16-bit quantity from host to network byte order
ntohl(val)	convert 32-bit quantity from network to host byte order
ntohs(val)	convert 16-bit quantity from network to host byte order

Table 3-1. C Run-Time Routines

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address (called the network byte order). Addresses supplied to system calls must be in network byte order; values returned by the system are also have this ordering. Because machines differ in the internal representation of integers, examining an address as returned by *getsockname*(2) or *getservbyname*(3N) may result in a misinterpretation. To use the number, it is necessary to use the routine *ntohs* (for *network to host: short*) to convert the number from the network representation to the host's representation, for example,

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines that have 'big-endian' byte ordering, such as the IRIS-4D, the *ntohs* is a null operation. On others with 'little-endian' ordering, such as the VAX, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called *htons*; the *ntohl* and *htonl* routines exist for long integers. Any protocol that transfers integer data between machines with different byte orders should use these routines. The library routines that return network addresses and ports provide them in network order so that they can simply be copied into the structures provided to the system.

3.4 Client/Server Model

The most commonly-used paradigm in constructing distributed applications is the client/server model. In this scheme, client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server. (See Section 3.2 for details.) This section examines the interactions between client and server, and considers some of the problems in developing client and server applications.

The client and server require a well-known set of conventions before service can be rendered (and accepted). This set of conventions comprises a protocol that must be implemented at both ends of a connection. The protocol can be symmetric or asymmetric. In a symmetric protocol, either side can play the master or slave roles. In an asymmetric protocol, one side is always the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the protocol is symmetric or asymmetric, when it accesses a service there is a "server process" and a "client process."

A server process normally listens at a well-known address for service requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up" and services the client, performing actions the client requests.

Alternative schemes that use a service server can eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers in 4.3BSD-based systems, this scheme has been implemented via *inetd*, the so called "internet super-server." The *inetd* daemon listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which *inetd* is listening, *inetd* executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as *inetd* has played any part in the connection. The *inetd* daemon is described in more detail in Section 3.5.

3.4.1 Servers

Most servers are accessed at well-known Internet addresses. The remote login server's main loop takes the form shown in the following example.

```
main(argc, argv)
    int argc;
    char *argv[];
{
    int f;
    struct sockaddr_in from;
    struct servent *sp;

    sp = getservbyname("login", "tcp");
    if (sp == NULL) {
        fprintf(stderr,
            "rlogind: tcp/login: unknown service\n");
        exit(1);
    }
    ...
#ifndef DEBUG
    /* Disassociate server from controlling terminal */
    ...
#endif
    /* Restricted port -- see Section 3.5.7 */
    sin.sin_port = sp->s_port;
    ...
    f = socket(AF_INET, SOCK_STREAM, 0);
    ...
    if (bind(f, (struct sockaddr *) &sin,
                sizeof (sin)) < 0) {
        syslog(LOG_ERR, "rlogind: bind: %m");
        exit(1);
    }
    ...
    listen(f, 5);
    for (;;) {
        int g, len = sizeof (from);
        g = accept(f, (struct sockaddr *) &from, &len);
        if (g < 0) {
            if (errno != EINTR) {
                syslog(LOG_ERR, "rlogind: accept: %m");
            }
            continue;
        }
```

```
            if (fork() == 0) {
                close(f);
                doit(g, &from);
            }
            close(g);
        }
    }
```

The first step taken by the server is to look up its service definition:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
    fprintf(stderr,
        "rlogind: tcp/login: unknown service\n");
    exit(1);
}
```

The result of the *getservbyname* call is used in later portions of the code to define the well-known Internet port where the server listens for service requests (indicated by a connection).

The second step taken by the server is to disassociate from the controlling terminal of its invoker:

```
for (i = getdtablesize(); i >= 0; i--) {
    (void) close(i);
}
open("/", O_RDONLY);
dup2(0, 1);
dup2(0, 2);

i = open("/dev/tty", O_RDWR);
if (i >= 0) {
    ioctl(i, TIOCNOTTY, 0);
    close(i);
}
```

This step protects the server from receiving signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself, it can no longer send reports of errors to a terminal, and must log errors via *syslog*.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. Note that the remote login server listens at a restricted port number, and must therefore be run with a user-id of root. This concept of a "restricted port number" is specific to 4.3BSD-based systems, and is covered in Section 3.5.

The main body of the loop is shown below:

```
for (;;) {
    int g, len = sizeof (from);

    g = accept(f, (struct sockaddr *)&from, &len);
    if (g < 0) {
        if (errno != EINTR) {
            syslog(LOG_ERR, "rlogind: accept: %m");
        }
        continue;
    }

    if (fork() == 0) {    /* Child */
        close(f);
        doit(g, &from);
    }
    close(g);             /* Parent */
}
```

An *accept* call blocks the server until a client requests service. This call could return a failure status if interrupted by a signal such as SIGCHLD. Therefore, the return value from *accept* is checked to insure a connection has actually been established, and an error report is logged via *syslog* if an error has occurred.

With a connection established, the server then forks a child process and invokes the main body of the remote login protocol processing. Note how the socket used by the parent for queuing connection requests is closed in the child, while the socket created as a result of the *accept* is closed in the parent. The address of the client is also handed to the *doit* routine because the routine requires it in authenticating clients.

3.4.2 Clients

The client side of the remote login service was shown earlier in Figure 3-1. The separate, asymmetric roles of the client and server show clearly in the code. The server is a passive entity, listening for client connections, while the client process is an active entity, initiating a connection when invoked.

Consider the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

```
sp = getservbyname("login", "tcp");
if (sp == NULL) {
    fprintf(stderr, "rlogin: tcp/login: unknown service\n");
    exit(1);
}
```

Next, the *gethostbyname* call looks up the destination host.

```
hp = gethostbyname(argv[1]);
if (hp == NULL) {
    fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
    exit(2);
}
```

Next, a connection is established to the server at the requested host and the remote login protocol is started. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number of the login process on the foreign host:

```
bzero((char *)&server, sizeof (server));
bcopy(hp->h_addr, (char *) &server.sin_addr, hp->h_length);
server.sin_family = hp->h_addrtype;
server.sin_port = sp->s_port;
```

A socket is created, and a connection initiated. Note that *connect* implicitly performs a *bind* call, since *s* is unbound.

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
    perror("rlogin: socket");
    exit(3);
}
 ...

if (connect(s, (struct sockaddr *) &server,
        sizeof (server)) < 0) {
    perror("rlogin: connect");
    exit(4);
}
```

3.4.3 Connectionless Servers

While connection-based services are the norm, some services are based on the use of datagram sockets. The "rwho" service is an example. It provides users with status information for hosts connected to a local area network. This service is predicated on the ability to *broadcast* information to all hosts connected to a particular network.

A user on any machine running the rwho server can find out the current status of a machine with the *ruptime*(1C) program. The output generated is illustrated in Figure 3-2.

```
dali       up   2+06:28,   9 users, load 1.04, 1.20, 1.65
renoir   down   0:24
miro       up   3+06:18,   0 users, load 0.03, 0.03, 0.05
monet      up   1+00:43,   2 users, load 0.22, 0.09, 0.07
```

Figure 3-2. Ruptime Output

Status information for each host is periodically broadcast by rwho server processes on each machine. The same server process also receives the status information and uses it to update a database. This database is then interpreted to generate the status information for each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

The use of broadcast for such a task is inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and frequently used, the expense of periodic broadcasts outweighs the simplicity. However, on a very small network, (for example, dedicated to a computation engine and several display engines) broadcast works well because all services are universal.

Multicasting is an alternative to broadcasting. Setting up multicast sockets is described in Section 3.5.

The *rwho* server, in a simplified form, is shown in the following example. The server performs two separate tasks. The first task is to receive status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are interrogated to make sure they were sent by another rwho server process. They are then time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for

an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server can be down while a host is actually up.

```
main()
{
    ...
    sp = getservbyname("who", "udp");
    sin.sin_addr.s_addr = htonl(INADDR_ANY);
    sin.sin_port = sp->s_port;
    ...

    s = socket(AF_INET, SOCK_DGRAM, 0);
    ...

    on = 1;
    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST,
                &on, sizeof(on)) < 0) {
        syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
        exit(1);
    }
    bind(s, (struct sockaddr *) &sin, sizeof (sin));
    ...

    signal(SIGALRM, onalrm);
    onalrm();
    for (;;) {
        struct whod wd;
        int cc, whod, len = sizeof (from);

        cc = recvfrom(s, (char *)&wd,
                sizeof (struct whod), 0,
                (struct sockaddr *)&from, &len);
        if (cc <= 0) {
            if (cc < 0 && errno != EINTR) {
                syslog(LOG_ERR, "rwhod: recv: %m");
            }
            continue;
        }
        if (from.sin_port != sp->s_port) {
            syslog(LOG_ERR, "rwhod: %d: bad from port",
                    ntohs(from.sin_port));
            continue;
        }
        ...

        if (!verify(wd.wd_hostname)) {
            syslog(LOG_ERR,
                    "rwhod: malformed host name from %x",
                    ntohl(from.sin_addr.s_addr));
            continue;
        }
```

```
            (void) sprintf(path, "%s/whod.%s",
                    RWHODIR, wd.wd_hostname);
            whod = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
            ...
            /* undo header byte swapping before writing to file */
            wd.wd_sendtime = ntohl(wd.wd_sendtime);
            ...

            (void) time(&wd.wd_recvtime);
            (void) write(whod, (char *)&wd, cc);
            (void) close(whod);
        }
    }
```

The second task performed by the server is to supply host status information. This involves periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other rwho servers to hear. The supply function is triggered by a timer and runs off a signal. Deciding where to transmit the resultant packet is somewhat problematical.

Status information must be broadcast on the local network. For networks that do not support broadcast, another scheme must be used. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This requires some bootstrapping information, because a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbors and thus never will receive, or send, any status information. This problem also occurs in the routing table management process in propagating routing status information. The standard solution is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information can then propagate through a neighbor to hosts that are not directly neighbors.

If the server is able to support networks that provide a broadcast capability, as well as those that do not, then networks with an arbitrary topology can share status information. However, "loops" can cause problems. That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

Software operating in a distributed environment should not have any site-dependent information compiled into it. This requires a separate copy of the server at each host and makes maintenance difficult. The 4.3BSD model

attempts to isolate host-specific information from applications by providing system calls that return the necessary information. An example of such a call is *gethostname*(2), which returns the host's "official" name. Also, an *ioctl* call can find the collection of networks to which a host is directly connected. Furthermore, a local network broadcasting mechanism has been implemented at the socket level. Combining these features allows a process to broadcast on any directly connected local network that supports the notion of broadcasting in a site-independent manner. This allows the system to decide how to propagate status information in the case of *rwho*, or more generally in broadcasting. Such status information is broadcast to connected networks at the socket level, where the connected networks have been obtained via the appropriate *ioctl* calls. The specifics of such broadcastings are covered in the next section.

3.5 Advanced Topics

For most users of sockets, the mechanisms already described will suffice in constructing distributed applications. However, you might need to use some of the more advanced features described in this section.

3.5.1 Out-of-Band Data

Stream sockets can accommodate "out-of-band" data. Out-of-band data are transmitted on a logically independent transmission channel associated with each pair of connected stream sockets. Out-of-band data are delivered to the user independently of normal data. For stream sockets, the out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message can contain at least one byte of data, and at least one message can be pending delivery to the user at any one time.

For communications protocols that support only in-band signaling (i.e., the urgent data are delivered in sequence with the normal data), the system extracts the data from the normal data stream and stores them separately. This allows you to choose between receiving the urgent data in sequence and receiving them out of sequence, without having to buffer all the intervening data.

It is possible to "peek" (via MSG_PEEK) at out-of-band data. If the socket has a process group, SIGURG is generated when the protocol is notified of its existence. A process can set the process group or process ID to be informed by SIGURG via the appropriate *fcntl* call, as described below for SIGIO. If multiple sockets can have out-of-band data awaiting delivery, a *select* call for exceptional conditions can be used to determine those sockets with such data pending. Neither the signal nor the *select* indicate the actual arrival of the out-of-band data, but only notification of pending data.

In addition to the information passed, a logical mark is placed in the data stream to indicate the point at which the out-of-band data were sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushes pending output from the remote process(es), all data up to the mark in the data stream are discarded.

To send an out-of-band message, the MSG_OOB flag is supplied to a *send* or *sendto* calls. To receive out-of-band data, MSG_OOB should be indicated when performing a *recvfrom* or *recv* call. To find out if the read pointer is currently pointing at the mark in the data stream, use the SIOCATMARK ioctl:

```
int yes;
ioctl(s, SIOCATMARK, &yes);
```

If the value *yes* is a 1 on return, the next *read* will return data after the mark. Otherwise (assuming out-of-band data have arrived), the next *read* will provide data sent by the client prior to transmission of the out-of-band signal. Figure 3-3 shows the routine used in the remote login process to flush output on receipt of an interrupt or quit signal. It reads the normal data up to the mark (to discard them), then reads the out-of-band byte.

A process can also read at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data and only sends notification of their presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a *recv* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, there may be so

much in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data for the urgent data to be delivered.

```
#include <stdio.h>
#include <termios.h>                /* POSIX-style */
#include <sys/ioctl.h>
#include <sys/socket.h>

oob()
{
    int  mark;
    char waste[BUFSIZ];

    /* Flush local terminal output */
    tcflush(1, TCOFLUSH);
    for (;;) {
        if (ioctl(rem, SIOCATMARK, &mark) < 0) {
            perror("ioctl");
            break;
        }
        if (mark)
            break;
        (void) read(rem, waste, sizeof (waste));
    }
    if (recv(rem, &mark, 1, MSG_OOB) < 0) {
        perror("recv");
        ...
    }
    ...
}
```

Figure 3-3. Flushing Terminal I/O on Receipt of Out-of-Band Data

Certain programs that use multiple bytes of urgent data and must handle multiple urgent signals (e.g., *telnet* (1C)) need to retain the position of urgent data within the stream. This treatment is available as a socket-level option, SO_OOBINLINE (see *setsockopt* (2) for usage). With this option, the position of urgent data (the "mark") is retained, but the urgent data immediately follow the mark within the normal data stream returned without the MSG_OOB flag. Reception of multiple urgent indications causes the mark to move, but no out-of-band data are lost.

3.5.2 Non-Blocking Sockets

Programs that cannot wait for a socket operation to be completed should use non-blocking sockets. I/O requests on non-blocking sockets return with an error if the request cannot be satisfied immediately.

Once a socket has been created via the *socket* call, it can be marked as non-blocking by *fcntl* as follows:

```
#include <fcntl.h>
 ...
int      s;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
 ...
if (fcntl(s, F_SETFL, FNDELAY) < 0)
    perror("fcntl F_SETFL, FNDELAY");
    exit(1);
}
 ...
```

When performing non-blocking I/O on sockets, check for the error EWOULDBLOCK (stored in the global variable *errno*). This occurs when an operation would normally block, but the socket it was performed on is non-blocking. In particular, *accept*, *connect*, *send*, *recv*, *read*, and *write* can all return EWOULDBLOCK, and processes should be prepared to deal with such return codes. If an operation such as a *send* cannot be completed, but partial writes are sensible (for example, when using a stream socket), the data that can be sent immediately are processed, and the return value indicates the amount actually sent.

3.5.3 Interrupt-Driven Socket I/O

The SIGIO signal allows a process to be notified when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps:

1. The process must use the *signal* call to set up a SIGIO signal handler.
2. The process must set the process ID or process group ID (see the next subsection) to receive notification of pending input either to its own process ID or to the process group ID of its process group (the default process group of a socket is group zero). To do this, the process uses *fcntl*.

3. The process uses another *fcntl* call to enable asynchronous notification of pending I/O requests. Figure 3-4 shows sample code to allow a process to receive information on pending I/O requests as they occur for a socket *s*. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#ifdef sgi
#define _BSD_SIGNALS
#endif
#include <signal.h>
#include <fcntl.h>
 ...
int     io_handler();
 ...
main()
{
    signal(SIGIO, io_handler);

    /*
     * Set the process receiving SIGIO/SIGURG
     * signals to us
     */

    if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
    }

    /* Allow receipt of asynchronous I/O signals */

    if (fcntl(s, F_SETFL, FASYNC) < 0) {
        perror("fcntl F_SETFL, FASYNC");
        exit(1);
    }
}

io_handler()
{
     ...
}
```

Figure 3-4. Use of Asynchronous Notification of I/O Requests

3.5.4 Signals and Process Groups

Due to the existence of the SIGURG and SIGIO signals, each socket has an associated process number, just as is done for terminals. This value is initialized to zero, but can be redefined at a later time with the F_SETOWN *fcntl*, as was done in the code above for SIGIO. To set the socket's process ID for signals, positive arguments should be given to the *fcntl* call. To set the socket's process group for signals, negative arguments should be passed to *fcntl*. Note that the process number indicates either the associated process ID or the associated process group; it is impossible to specify both at the same time. A similar *fcntl*, F_GETOWN, is available for determining the current process number of a socket.

SIGCHLD is another signal that is useful when constructing server processes. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to "reap" child processes that have exited, without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Section 3.4.1 can be augmented as shown in Figure 3-5.

```
#ifdef sgi
#define _BSD_SIGNALS
#endif
#include <signal.h>

int reaper();
 ...
main()
{
    ...
    signal(SIGCHLD, reaper);
    listen(f, 5);
    for (;;) {
        int g, len = sizeof (from);

        g = accept(f, (struct sockaddr *)&from, &len,);
        if (g < 0) {
            if (errno != EINTR)
                syslog(LOG_ERR, "rlogind: accept: %m");
            continue;
        }
        ...
    }
}
#include <sys/wait.h>
reaper()
{
    union wait status;

    while (wait3(&status, WNOHANG, 0) > 0) {
            ;    /* no-op */
    }
}
```

Figure 3-5. Use of the SIGCHLD Signal

If the parent server process fails to reap its children, a large number of "zombie" processes can be created.

3.5.5 Pseudo-Terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicate over the network through a *pseudo-terminal.* A pseudo-terminal is actually a pair of devices, master and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal are supplied as input to a process reading from the

master side, while data written on the master side are processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection. The slave side appears as a normal terminal to any process reading from or writing to it.

For example, the remote login server uses pseudo-terminals for remote login sessions. A user logging in to a machine across the network gets a shell with a slave pseudo-terminal as standard input, output, and error. The server process then handles the communication between the programs invoked by the remote shell and the user's local client process. When a user sends a character that causes a remote machine to flush terminal output, the pseudo-terminal generates a control message for the server process. The server then sends an out-of-band message to the client process to signal a flush of data at the real terminal and on the intervening data buffered in the network.

Under IRIX, the name of the slave side of a pseudo-terminal has the syntax:

```
/dev/ttyqx
```

In this syntax, *x* is a number in the range 0 through 99. The master side of a pseudo-terminal is the generic device */dev/ptc*.

Creating a pair of master and slave pseudo-terminals is straightforward. The master half of a pseudo-terminal is opened first. The slave side of the pseudo-terminal is then opened and is set to the proper terminal modes if necessary. The process then *fork*s. The child closes the master side of the pseudo-terminal and *exec*s the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Figure 3-6 illustrates sample code making use of pseudo-terminals. This code assumes that a connection on a socket *s* exists, connected to a peer that wants a service of some kind, and that the process has disassociated itself from any previously controlling terminal.

```
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <fcntl.h>
#include <syslog.h>

int master, slave;
struct stat stb;
char line[sizeof("/dev/ttyqxxx")];

master = open("/dev/ptc", O_RDWR | O_NDELAY);
if (master < 0 || fstat(master, &stb) < 0) {
    syslog(LOG_ERR, "All network ports in use");
    exit(1);
}
sprintf(line, "/dev/ttyq%d", minor(stb.st_rdev));

/*
 * Put in separate process group, disassociate
 * controlling terminal.
 */
setpgrp();       /* SYSV version, not BSD */

slave = open(line, O_RDWR);      /* Open slave side */
if (slave < 0) {
    syslog(LOG_ERR, "Cannot open slave pty %s", line);
    exit(1);
}

pid = fork();
if (pid < 0) {
    syslog(LOG_ERR, "fork: %m");
    exit(1);
}
if (pid > 0) {   /* Parent */
    close(slave);
    ...
} else {                 /* Child */
    close(f);
    close(master);
    dup2(slave, 0);
    dup2(slave, 1);
    dup2(slave, 2);
    if (slave > 2)
        (void) close(slave);
    ...
}
```

Figure 3-6. Creation and Use of a Pseudo-Terminal on IRIX

3.5.6 Selecting Protocols

If the third argument to the *socket* call is 0, *socket* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using raw sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument can be important for setting up demultiplexing. For example, raw sockets in the Internet family can be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified.

To obtain a particular protocol, determine the protocol number as defined within the communication domain. For the Internet domain, you can use one of the library routines discussed in Section 3.3, such as *getprotobyname*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET, SOCK_STREAM, pp->p_proto);
```

This would result in a socket *s* using a stream-based connection, but with protocol type of "newtcp" instead of the default "tcp."

3.5.7 Address Binding

Binding addresses to sockets in the Internet domain can be fairly complex. These associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the *bind* system call, a process can specify half of an association, the <local address, local port> part, while the *connect* and *accept* primitives are used to complete a socket's association by specifying the <foreign address, foreign port> part. Since the association is created in two steps, the association uniqueness requirement could be violated unless care is taken. Furthermore, user programs will not always know proper values to use for the local address and local port, since a host can reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain, a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in *<netinet/in.h>*), the system interprets the address as "any valid address."

For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

```
#include <sys/types.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Sockets with wildcarded local addresses can receive messages directed to the specified port number and sent to any of the possible addresses assigned to a host. For example, if a host has addresses 128.32.0.4 and 10.0.0.78, and a socket is bound as above, the process will be able to accept connection requests that are addressed to 128.32.0.4 or 10.0.0.78. For a server process to allow only hosts on a given network connect to it, it would bind the address of the host on the appropriate network.

Similarly, a local port can be left unspecified (specified as zero), in which case the system selects an appropriate port number for it. For example, to bind a specific local address to a socket, but to leave the local port number unspecified, use the following code:

```
hp = gethostbyname(hostname);
if (hp == NULL) {
    ...
}
bcopy(hp->h_addr, (char *) sin.sin_addr, hp->h_length);
sin.sin_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The system selects the local port number based on two criteria. The first criterion is that, on 4.3BSD systems, Internet ports between 512 and 1023 (IPPORT_RESERVED – 1) are reserved for privileged users; Internet ports above IPPORT_USERRESERVED (5000) are reserved for non-privileged servers.

The second criterion is that the port number is not currently bound to some other socket. In order to find a free Internet port number in the privileged range, the *rresvport* library routine can be used as follows to return a stream socket with a privileged port number:

```
int lport = IPPORT_RESERVED - 1;
int s;
 ...
s = rresvport(&lport);
if (s < 0) {
    if (errno == EAGAIN)
        fprintf(stderr, "socket: all ports in use");
    else
        perror("rresvport: socket");
    ...
}
```

The restriction on allocating ports allows processes executing in a "secure" environment to perform authentication based on the originating address and port number. For example, the *rlogin*(1C) command allows users to log in across a network without being asked for a password, if two conditions hold: First, the name of the system the user is logging in from is in the file */etc/hosts.equiv* on the system being logged in to (or the system name and the user name are in the user's *.rhosts* file in the user's home directory). Second, the user's rlogin process is coming from a privileged port on the machine the user is logging in from. The port number and network address of the machine the user is logging in from can be determined either by the *from* result of the *accept* call, or from the *getpeername* call.

The algorithm used by the system to select port numbers can be unsuitable for an application. This is because the algorithm creates associations in a two step process. For example, the Internet file transfer protocol, FTP, specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. The system disallows binding the same local address and port number to a socket if a previous data connection's socket still existed. To override the default port selection algorithm, the following option call must be performed before address binding:

```
 ...
int     on = 1;
 ...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

With the above call, local addresses that are already in use can be bound. This does not violate the uniqueness requirement, as the system still checks at connect time to be sure any other sockets with the same local address and port do not have the same foreign address and port. If the association already exists, the error EADDRINUSE is returned.

3.5.8 Socket Options

You can use the *setsockopt* and *getsockopt* system calls to set and get a number of options on sockets. These options include marking a socket for broadcasting, not to route, to linger on close, etc.

The general forms of the calls are:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The parameters to the calls are as follows:

- *s* is the socket on which the option is to be applied.
- *level* specifies the protocol layer on which the option is to be applied; in most cases this is the "socket level," indicated by the symbolic constant SOL_SOCKET, defined in *<sys/socket.h>*.
- *optname* specifies the actual option, a symbolic constant also defined in *<sys/socket.h>*.
- *optval* points to the value of the option (in most cases, whether the option is to be turned on or off).
- *optlen* points to the length of the value of the option. For *getsockopt*, *optlen* is a value-result parameter, initially set to the size of the storage area pointed to by *optval*, and modified upon return to indicate the actual amount of storage used.

For example, it is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket. Programs under *inetd* (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the *getsockopt* call.

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);

if (getsockopt(s, SOL_SOCKET, SO_TYPE,
        (char *) &type, &size) < 0) {
    perror("getsockopt");
    ...
}
```

After the *getsockopt* call, *type* will be set to the value of the socket type, as defined in *<sys/socket.h>*. For example, if the socket were a datagram socket, *type* would have the value corresponding to SOCK_DGRAM.

3.5.9 Inetd

IRIX provides the daemon *inetd*(1M), the so-called "internet super-server." Having one daemon listen for requests for many daemons instead of having each daemon listen for its own requests reduces the number of idle daemons and simplies their implementation. *Inetd* handles three types of services: standard, RPC and TCPMUX. A standard service has a well-known port assigned to it and is listed in */etc/services* or the Yellow Pages *services* map (see *services*(4)); it may be a service that implements an official Internet standard or is a BSD Unix-specific service. RPC services use the Sun RPC calls as the transport; such services are listed in */etc/rpc* or the Yellow Pages *rpc* map (see *rpc*(4)). TCPMUX services are nonstandard and do not have a well-known port assigned to them. They are invoked from *inetd* when a program connects to the "tcpmux" well-known port and specifies the service name. This is useful for adding locally developed servers.

The *inetd* daemon, which is described in more detail in *inetd*(1M), is invoked at boot time. It examines the file */usr/etc/inetd.conf* to determine the servers it will listen for. Once this information has been read and a pristine environment created, *inetd* proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

The *inetd* daemon performs a *select* on these sockets for *read* availability, waiting for a process to request a connection to the service corresponding to that socket. The *inetd* daemon then performs an *accept* on the socket in question, *fork*s, *dup*s the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and *exec*s the appropriate server.

Servers making use of *inetd* are considerably simplified, as *inetd* takes care of most of the IPC work required in establishing a connection. The server invoked by *inetd* expects the socket connected to its client on file descriptors 0 and 1, and can immediately perform any operations such as *read*, *write*, *send*, or *recv*. Indeed, servers can use buffered I/O as provided by the "stdio" conventions, as long as as they use *fflush* when appropriate. However, for server programs that handle multiple services or protocols, *inetd* allocates socket descriptors to protocols based on lexicographic order of service and protocol name. For example, the RPC mount daemon, rpc.mountd has two entries in inetd.conf for its TCP and UDP ports. When invoked by *inetd*, the TCP socket is on descriptor 0, and UDP on 1.

When writing servers under *inetd*, you can use the *getpeername* call to return the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in "dot notation" (e.g., "128.32.0.4") of a client, you might use the following code:

```
struct sockaddr_in name;
int namelen = sizeof (name);
 ...
if (getpeername(0, (struct sockaddr *)&name,
        &namelen) < 0) {

    syslog(LOG_ERR, "getpeername: %m");
    exit(1);
} else {
    syslog(LOG_INFO, "Connection from %s",
                inet_ntoa(name.sin_addr));
}
```

While the *getpeername* call is especially useful when writing programs to run with *inetd*, it can be used by stand-alone servers.

Standard TCP services are assigned unique well-known port numbers in the range of 0 to 255. These ports are of limited number and are typically only assigned to official Internet protocols. The TCPMUX service, as described in RFC-1078, allows you to add locally-developed protocols without needing an official TCP port assignment. The protocol used by TCPMUX is simple: a TCP client connects to a foreign host on TCP port 1. It sends the

service name followed by a carriage-return line-feed <CRLF>. The server replies with a single character indicating positive ("+") or negative ("–") acknowledgment, immediately followed by an optional message of explanation, terminated with a <CRLF>. If the reply was positive, the selected protocol begins; otherwise the connection is closed. In IRIX, the TCPMUX service is built into *inetd*, that is, *inetd* listens on TCP port 1 for requests for TCPMUX services listed in *inetd.conf*.

The following is an example TCPMUX server and its *inetd.conf* entry. More sophisticated servers may want to do additional processing before returning the positive or negative acknowledgement.

```
#include <sys/types.h>
#include <stdio.h>

main()
{
        time_t t;

        printf("+Go\r\n");
        fflush(stdout);
        time(&t);
        printf("%d = %s", t, ctime(&t));
        fflush(stdout);
}
```

The *inetd.conf* entry is:

```
tcpmux/current_time stream tcp nowait guest /d/curtime curtime
```

Here's the portion of the client code that handles the TCPMUX handshake:

```
char line[BUFSIZ];
FILE *fp;
 ...

/* Use stdio for reading data from the server */
fp = fdopen(sock, "r");
if (fp == NULL) {
    fprintf(stderr, "Can't create file pointer0);
    exit(1);
}

/* Send service request */
sprintf(line, "%s\r\n", "current_time");
if (write(sock, line, strlen(line)) < 0) {
    perror("write");
    exit(1);
}
```

```
/* Get ACK/NAK response from the server */
if (fgets(line, sizeof(line), fp) == NULL) {
    if (feof(fp)) {
        die();
    } else {
        fprintf(stderr, "Error reading response0);
        exit(1);
    }
}

/* Delete <CR> */       ')) != NULL) {
if ((lp = index(line, '
    *lp = ' ';
}

switch (line[0]) {
    case '+':
            printf("Got ACK: %s0, &line[1]);
            break;
    case '-':
            printf("Got NAK: %s0, &line[1]);
            exit(0);
    default:
            printf("Got unknown response: %s0, line);
            exit(1);
}

/* Get rest of data from the server */
while ((fgets(line, sizeof(line), fp)) != NULL) {
    fputs(line, stdout);
}
```

3.5.10 Broadcasting

By using a datagram socket, you can send broadcast packets on many networks supported by the system. The network itself must support broadcast; the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network, since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets explicitly marked to allow broadcasting. Broadcast is typically used for one of two reasons: to find a resource on a local network without prior knowledge of its address, or to send information to all accessible neighbors.

Multicasting is an alternative to broadcasting. Setting up multicast sockets is described in the next section.

To send a broadcast message, create a datagram socket:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

Mark the socket to allow broadcasting:

```
int on = 1;

setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

Bind a port number to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

The destination address of the broadcast message depends on the network(s). The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in *<netinet/in.h>*. Determining the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, IRIX provides a method of retrieving this information from the system data structures.

The SIOCGIFCONF *ioctl* call returns the interface configuration of a host in the form of a single *ifconf* structure. This structure contains a data area that is made up of an array of of *ifreq* structures, one for each network interface to which the host is connected. These structures are defined in *<net/if.h>* as shown in the following example.

```
struct ifconf {
    int       ifc_len;    /* size of associated buffer */
    union {
        caddr_t  ifcu_buf;
        struct   ifreq *ifcu_req;
    } ifc_ifcu;
};
/* Buffer address */
#define ifc_buf   ifc_ifcu.ifcu_buf

/* Array of structures returned */
#define ifc_req   ifc_ifcu.ifcu_req

#define IFNAMSIZ 16

struct ifreq {
    /* Interface name, e.g. "enp0" */
    char ifr_name[IFNAMSIZ];

    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        short    ifru_flags;
        int      ifru_metric;
        caddr_t  ifru_data;
#ifdef sgi
        struct   ifstats ifru_stats;
#endif
    } ifr_ifru;
};

/* Address */
#define ifr_addr      ifr_ifru.ifru_addr

/* Other end of p-to-p link */
#define ifr_dstaddr   ifr_ifru.ifru_dstaddr

/* Broadcast address */
#define ifr_broadaddr ifr_ifru.ifru_broadaddr

/* Flags */
#define ifr_flags     ifr_ifru.ifru_flags

/* Metri */
#define ifr_metric    ifr_ifru.ifru_metric

/* For use by interface */
#define ifr_data      ifr_ifru.ifru_data
```

The following call obtains the interface configuration:

```
struct ifconf ifc;
char buf[BUFSIZ];

ifc.ifc_len = sizeof (buf);
ifc.ifc_buf = buf;
if (ioctl(s, SIOCGIFCONF, (char *) &ifc) < 0) {
    ...
}
```

After this call, *buf* will contain one *ifreq* structure for each network to which the host is connected, and *ifc.ifc_len* will have been modified to reflect the number of bytes used by the *ifreq* structures.

Each structure has an associated set of "interface flags" that tell whether the network corresponding to that interface is up or down, point-to-point or broadcast, etc. The SIOCGIFFLAGS *ioctl* retrieves these flags for an interface specified by an *ifreq* structure as follows:

```
struct ifreq *ifr;
struct sockaddr dst;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq);
     --n >= 0;
     ifr++) {

    /*
     * Be careful not to use an interface
     * devoted to an address family other than
     * the one intended.
     */
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
        ...
    }
    /*
     * Skip boring cases.
     */
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
        (ifr->ifr_flags & IFF_LOOPBACK) ||
        (ifr->ifr_flags &
            (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0) {
        continue;
    }
```

Once you retrieve the flags, retrieve the broadcast address. For broadcast networks, this is done via the SIOCGIFBRDADDR *ioctl*. For point-to-point networks, the address of the destination host is obtained with SIOCGIFDSTADDR.

```
if (ifr->ifr_flags & IFF_POINTTOPOINT) {
    if (ioctl(s, SIOCGIFDSTADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_dstaddr, (char *) &dst,
                sizeof (ifr->ifr_dstaddr));

} else if (ifr->ifr_flags & IFF_BROADCAST) {
    if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
        ...
    }
    bcopy((char *) ifr->ifr_broadaddr, (char *) &dst,
                sizeof (ifr->ifr_broadaddr));
}
```

After the appropriate *ioctls* get the broadcast or destination address (now in *dst*), use the *sendto* call:

```
    sendto(s, buf, buflen, 0,
                (struct sockaddr *)&dst, sizeof (dst));
}
```

In the above loop, one *sendto* occurs for every interface the host is connected to that supports broadcast or point-to-point addressing. For a process to send only broadcast messages on a given network, use code similar to that outlined above, but the loop would need to find the correct destination address.

Received broadcast messages contain the sender's address and port, as datagram sockets are bound before a message is allowed to go out.

3.5.11 Multicasting

IP multicasting is the transmission of an IP datagram to a "host group", a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same "best-efforts" reliability as regular unicast IP datagrams, i.e., the datagram is not guaranteed to arrive intact at all members of the destination group or in the same order relative to other datagrams.

The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time. A host need not be a member of a group to send datagrams to it.

A host group may be permanent or transient. A permanent group has a well-known, administratively assigned IP address. It is the address, not the membership of the group, that is permanent; at any time a permanent group may have any number of members, even zero. Those IP multicast addresses that are not reserved for permanent groups are available for dynamic assignment to transient groups which exist only as long as they have members.

In general, a host cannot assume that datagrams sent to any host group address will reach only the intended hosts, or that datagrams received as a member of a transient host group are intended for the recipient. Misdelivery must be detected at a level above IP, using higher-level identifiers or authentication tokens. Information transmitted to a host group address should be encrypted or governed by administrative routing controls if the sender is concerned about unwanted listeners.

Note: This RFC-1112 level-2 implementation of IP multicasting is experimental and subject to change in order to track future BSD UNIX releases. In particular, there may be changes in the way a process overrides the default interface for sending multicast datagrams and for joining multicast groups. This ability to override the default interface is intended mainly for routing demons; normal applications should not be concerned with specific interfaces.

IP multicasting is currently supported only on AF_INET sockets of type SOCK_DGRAM and SOCK_RAW, and only on subnetworks for which the interface driver has been modified to support multicasting. The standard Ethernet and SLIP interfaces on the IRIS-4D support multicasting. (Older versions of ENP-10 interfaces may require an upgrade — see Chapter 1 for details.)

The next subsections describe how to send and receive multicast datagrams.

Sending IP Multicast Datagrams

To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255 as the destination address in a *sendto*(2) call.

The definitions required for the multicast-related socket options are found in *<netinet/in.h>*. All IP addresses are passed in network byte-order.

By default, IP multicast datagrams are sent with a time-to-live (TTL) of 1, which prevents them from being forwarded beyond a single subnetwork. A new socket option allows the TTL for subsequent multicast datagrams to be set to any value from 0 to 255, in order to control the scope of the multicasts:

```
u_char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL,
                  &ttl, sizeof(ttl));
```

Multicast datagrams with a TTL of 0 will not be transmitted on any subnet, but may be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket (see below). Multicast datagrams with TTL greater than one may be delivered to more than one subnet if there are one or more multicast routers attached to the first-hop subnet. To provide meaningful scope control, the multicast routers support the notion of TTL "thresholds", which prevent datagrams with less than a certain TTL from traversing certain subnets. The thresholds enforce the following convention:

Scope	Initial TTL
restricted to the same host	0
restricted to the same subnet	1
restricted to the same site	32
restricted to the same region	64
restricted to the same continent	128
unrestricted	255

"Sites" and "regions" are not strictly defined, and sites may be further subdivided into smaller administrative units, as a local matter.

An application may choose an initial TTL other than the ones listed above. For example, an application might perform an "expanding-ring search" for a network resource by sending a multicast query, first with a TTL of 0, and

then with larger and larger TTLs, until a reply is received, perhaps using the TTL sequence 0, 1, 2, 4, 8, 16, 32.

The multicast router *mrouted*(1M), refuses to forward any multicast datagram with a destination address between 224.0.0.0 and 224.0.0.255, inclusive, regardless of its TTL. This range of addresses is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting.

The address 224.0.0.0 is guaranteed not to be assigned to any group, and 224.0.0.1 is assigned to the permanent group of all IP hosts (including gateways). This is used to address all multicast hosts on the directly connected network. There is no multicast address (or any other IP address) for all hosts on the total Internet. The addresses of other well-known, permanent groups are published in the "Assigned Numbers" RFC, which is available from the NIC.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. (If the host is also serving as a multicast router, a multicast may be *forwarded* to interfaces other than originating interface, provided that the TTL is greater than 1.) The default interface to be used for multicasting is the primary network interface on the system. A socket option is available to override the default for subsequent transmissions from a given socket:

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF,
                &addr, sizeof(addr));
```

where "addr" is the local IP address of the desired outgoing interface. An address of INADDR_ANY may be used to revert to the default interface. The local IP address of an interface can be obtained via the SIOCGIFCONF ioctl. To determine if an interface supports multicasting, fetch the interface flags via the SIOCGIFFLAGS ioctl and see if the IFF_MULTICAST flag is set. (Normal applications should not need to use this option; it is intended primarily for multicast routers and other system services specifically concerned with internet topology.) The SIOCGIFCONF and SIOCGIFFLAGS ioctls are described in the previous section.

If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether or not subsequent datagrams are looped back:

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP,
                  &loop, sizeof(loop));
```

where *loop* is set to 0 to disable loopback, and set to 1 to enable loopback. This option improves performance for applications that may have no more than one instance on a single host (such as a router demon), by eliminating the overhead of receiving their own transmissions. It should generally not be used by applications for which there may be more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time querying program).

A multicast datagram sent with an initial TTL greater than 1 may be delivered to the sending host on a different interface from that on which it was sent, if the host belongs to the destination group on that other interface. The loopback control option has no effect on such delivery.

Receiving IP Multicast Datagrams

Before a host can receive IP multicast datagrams, it must become a member of one or more IP multicast groups. A process can ask the host to join a multicast group by using the following socket option:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                  &mreq, sizeof(mreq))
```

where "mreq" is the following structure:

```
struct ip_mreq {
    struct in_addr imr_multiaddr; /* multicast group to join */
    struct in_addr imr_interface; /* interface to join on */
}
```

Every membership is associated with a single interface, and it is possible to join the same group on more than one interface. "imr_interface" should be INADDR_ANY to choose the default multicast interface, or one of the host's local addresses to choose a particular (multicast-capable) interface.

Up to IP_MAX_MEMBERSHIPS (currently 20) memberships may be added on a single socket.

To drop a membership, use:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                  &mreq, sizeof(mreq));
```

where "mreq" contains the same values as used to add the membership. The memberships associated with a socket are also dropped when the socket is closed or the process holding the socket is killed. However, more than one socket may claim a membership in a particular group, and the host will remain a member of that group until the last claim is dropped.

The memberships associated with a socket do not necessarily determine which datagrams are received on that socket. Incoming multicast packets are accepted by the kernel IP layer if any socket has claimed a membership in the destination group of the datagram; however, delivery of a multicast datagram to a particular socket is based on the destination port (or protocol type, for raw sockets), just as with unicast datagrams. To receive multicast datagrams sent to a particular port, it is necessary to bind to that local port, leaving the local address unspecified (i.e., INADDR_ANY).

More than one process may bind to the same SOCK_DGRAM UDP port if the *bind* call is preceded by:

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));
```

In this case, every incoming multicast or broadcast UDP datagram destined to the shared port is delivered to all sockets bound to the port. For backwards compatibility reasons, this does not apply to incoming unicast datagrams. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port. SOCK_RAW sockets do not require the SO_REUSEADDR option to share a single IP protocol type.

A final multicast-related extension is independent of IP: two new ioctls, SIOCADDMULTI and SIOCDELMULTI, are available to add or delete link-level (e.g., Ethernet) multicast addresses accepted by a particular interface. The address to be added or deleted is passed as a sockaddr structure of family AF_UNSPEC, within the standard ifreq structure.

These ioctls are for the use of protocols other than IP, and require superuser privileges. A link-level multicast address added via SIOCADDMULTI is not automatically deleted when the socket used to add it goes away; it must be explicitly deleted. It is inadvisable to delete a link-level address that may be in use by IP.

Sample Multicast Program

The following program sends or receives multicast packets. If invoked with one argument, it sends a packet containing the current time to an arbitrarily-chosen multicast group and UDP port. If invoked with no arguments, it receives and prints these packets. Start it as a sender on just one host and as a receiver on all the other hosts.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <stdio.h>

#define EXAMPLE_PORT     6000
#define EXAMPLE_GROUP    "224.0.0.250"

main(argc)
     int argc;
{
     struct sockaddr_in addr;
     int addrlen, fd, cnt;
     struct ip_mreq mreq;
     char message[50];

     fd = socket(AF_INET, SOCK_DGRAM, 0);
     if (fd < 0) {
          perror("socket");
          exit(1);
     }

     bzero(&addr, sizeof(addr));
     addr.sin_family = AF_INET;
     addr.sin_addr.s_addr = htonl(INADDR_ANY);
     addr.sin_port = htons(EXAMPLE_PORT);
     addrlen = sizeof(addr);
```

```
    if (argc > 1) {      /* Send */
        addr.sin_addr.s_addr = inet_addr(EXAMPLE_GROUP);
        while (1) {
            time_t t = time(0);
            sprintf(message, "time is %-24.24s", ctime(&t));
            cnt = sendto(fd, message, sizeof(message), 0,
                    &addr, addrlen);
            if (cnt < 0) {
                perror("sendto");
                exit(1);
            }
            sleep(5);
        }
    } else {             /* Receive */
        if (bind(fd, &addr, sizeof(addr)) < 0) {
            perror("bind");
            exit(1);
        }

        mreq.imr_multiaddr.s_addr = inet_addr(EXAMPLE_GROUP);
        mreq.imr_interface.s_addr = htonl(INADDR_ANY);
        if (setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                    &mreq, sizeof(mreq)) < 0) {
            perror("setsockopt mreq");
            exit(1);
        }

        while (1) {
            cnt = recvfrom(fd, message, sizeof(message), 0,
                        &addr, &addrlen);
            if (cnt < 0) {
                    perror("recvfrom");
                    exit(1);
            } else if (cnt == 0) {
                break;
            }
            printf("%s: message = \"%s\"\n",
                    inet_ntoa(addr.sin_addr), message);
        }
    }
}
```

4. RPC Programming

This chapter is written for programmers who want to write network applications using remote procedure calls, thus avoiding low-level system primitives based on sockets. This chapter is also for those who want to understand the RPC mechanisms usually hidden by the *rpcgen*(1) protocol compiler. The RPC language and *rpcgen* are described in Chapter 5. The RPC protocol is described in Chapter 8.

This chapter describes:

- the high, middle, and low layers of RPC
- RPC features such as broadcast, batching, authentication
- the entry points (routines) into the RPC system

To use this chapter, you must be familiar with the C programming language, and should have a working knowledge of network theory. For most applications, you can circumvent the need to cope with the details presented here by using *rpcgen.* In Chapter 5, "Generating XDR Routines" contains the complete source for a working RPC service—a remote directory listing service that uses *rpcgen* to generate XDR routines as well as client and server stubs.

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

4.1 Layers of RPC

Think of the RPC interface as being divided into three layers.

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

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

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

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

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

The Lowest Layer: The lowest layer does allow these details to be controlled by the programmer. Programs written at this level are also most efficient, but this is rarely an issue — since RPC clients and servers rarely generate heavy network loads. The lowest layer is used for more sophisticated applications that may want to alter the defaults of the routines. At this layer, you can explicitly manipulate sockets used for transmitting RPC messages. This level should be avoided if possible.

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

Programs that communicate over a network need a paradigm for communication. A low-level mechanism might send a signal on the arrival of incoming packets, causing a network signal handler to execute. A high-level mechanism would be the Ada *rendezvous*. The method described here is the Remote Procedure Call (RPC) paradigm, in which a client communicates with a server. In this process, the client first calls a procedure to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, sends back the reply, and the procedure call returns to the client.

A diagram of the RPC paradigm appears in Figure 4-1.

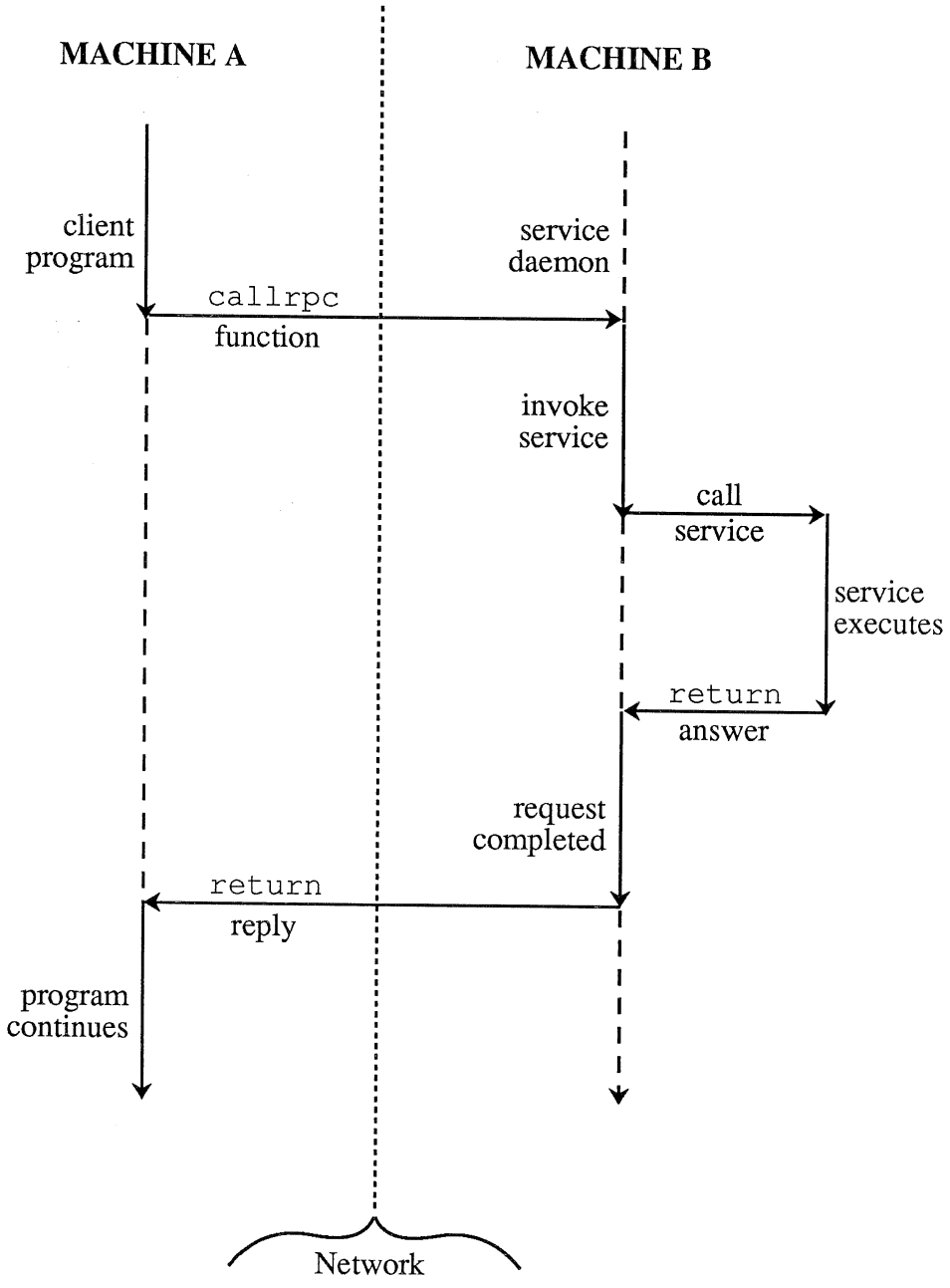


Figure 4-1. Network Communication with the Remote Procedure Call

4.2 Higher Layers of RPC

This section describes the highest and intermediate layers of RPC.

4.2.1 Highest Layer

Suppose you're writing a program that needs to know how many users are logged into a remote machine. You can do this by calling the RPC library routine *rnusers()* as shown in the following program fragment.

```
#include <stdio.h>

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

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

RPC library routines such as *rnusers()* are included in the C library *librpcsvc.a*. Thus, you can compile the program above with *cc*.

```
cc prog.c -lrpcsvc -lsun -o prog
```

(See the section on compiling BSD programs in Chapter 3 for other compiling hints.) Another library routine, *rstat()*, gathers remote performance statistics.

4.2.2 Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions: *callrpc()* and *registerrpc()*. Another way to get the number of remote users is:

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

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

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

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

The simplest way of making remote procedure calls is with the the RPC library routine *callrpc()*. It has eight parameters. The first is the name of the remote server machine. The next three parameters are the program, version, and procedure numbers—together they identify the procedure to be called. The fifth and sixth parameters are an XDR filter and an argument to

be encoded and passed to the remote procedure. The final two parameters are a filter for decoding the results returned by the remote procedure and a pointer to the place where the procedure's results are to be stored. Multiple arguments and results are handled by embedding them in structures. If *callrpc()* completes successfully, it returns zero; else it returns a nonzero value. The return codes (of type cast into an integer) are found in *<rpc/clnt.h>* .

Since data types may be represented differently on different machines, *callrpc()* needs both the type of the RPC argument, as well as a pointer to the argument itself (and similarly for the result). For RUSERSPROC_NUM, the return value is an `unsigned long`. So *callrpc()* has *xdr_u_long* as its first return parameter, which says that the result is of type `unsigned long`, and *&nusers* as its second return parameter, which is a pointer to where the long result will be placed. Since RUSERSPROC_NUM takes no argument, the argument parameter of *callrpc()* is *xdr_void.*

After trying several times to deliver a message, if *callrpc()* gets no answer, it returns with an error code. The delivery mechanism is the User Datagram Protocol (UDP). Methods for adjusting the number of retries or for using a different protocol require you to use the lower layer of the RPC library, discussed later in this chapter. The remote server procedure corresponding to the above might look like this:

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

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

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

Normally, a server registers all of the RPC calls it plans to handle, and then goes into an infinite loop waiting to service requests. In this example, there is only a single procedure to register, so the main body of the server would look like the following example.

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

char *nuser();

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

The *registerrpc()* routine establishes what C procedure corresponds to each RPC procedure number. The first three parameters, RUSERPROG, RUSERSVERS, and RUSERSPROC_NUM are the program, version, and procedure numbers of the remote procedure to be registered; *nuser* is the name of the C procedure implementing it; and *xdr_void* and *xdr_u_long* are the XDR filters for the remote procedure's arguments and results, respectively. (Multiple arguments or multiple results are passed as structures).

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

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

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

4.2.3 Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 (536870912) according to the chart that follows.

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

Sun Microsystems administers the first group of numbers. The second group of numbers is reserved for specific customer applications. This range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

To register a protocol specification, send a request by network mail to:

```
rpc@sun.com
```

or

```
sun!rpc
```

or write to:

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

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

You can find the RPC program numbers and protocol specifications of standard Sun RPC services in the include files in */usr/include/rpcsvc*. These services, however, constitute only a small subset of those that have been registered. Some of the current registered programs are listed in the table that follows.

RPC Number	Program	Description
100000	PMAPPROG	portmapper
100001	RSTATPROG	remote stats
100002	RUSERSPROG	remote users
100003	NFSPROG	nfs
100004	YPPROG	Yellow Pages
100005	MOUNTPROG	mount demon
100006	DBXPROG	remote dbx
100007	YPBINDPROG	yp binder
100008	WALLPROG	shutdown msg
100009	YPPASSWDPROG	yppasswd server
100010	ETHERSTATPROG	ether stats
100012	SPRAYPROG	spray packets
100017	REXECPROG	remote execution
100020	LOCKPROG	local lock manager
100021	NETLOCKPROG	network lock manager
100023	STATMON1PROG	status monitor 1
100024	STATMON2PROG	status monitor 2
100026	BOOTPARAMPROG	boot parameters service
100028	YPUPDATEPROG	yp update
100029	KEYSERVEPROG	key server
100036	PWDAUTHPROG	password authorization

Table 4-1. RPC Registered Programs

4.2.4 Passing Arbitrary Data Types

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

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

Note that the routine *xdr_string()* exists, but cannot be used with *callrpc()* and *registerrpc()*, which only pass two parameters to their XDR routines. *xdr_wrapstring()* has only two parameters, and is thus OK. It calls *xdr_string()*.

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

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

then you would call *callrpc* as

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

where *xdr_simple()* is written as:

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

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

An XDR routine returns nonzero (true in the sense of C) if it completes successfully, and zero otherwise. A complete description of XDR is in Chapter 7, so this section only gives a few examples of XDR implementation.

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

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

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

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

and make an RPC call such as:

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

with *xdr_varintarr()* defined as:

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

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

If the size of the array is known in advance, one can use *xdr_vector()*, which serializes fixed-length arrays.

```
int intarr[SIZE];

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

XDR always converts quantities to 4-byte multiples when serializing. Thus, if either of the examples above involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine

xdr_bytes(), which is like *xdr_array()* except that it packs characters; *xdr_bytes()* has four parameters, similar to the first four parameters of *xdr_array()*. For null-terminated strings, there is also the *xdr_string()* routine, which is the same as *xdr_bytes()* without the length parameter. On serializing it gets the string length from *strlen()*, and on deserializing it creates a null-terminated string.

Here is a final example that calls the previously written *xdr_simple()* as well as the built-in functions *xdr_string()* and *xdr_reference()*, which chases pointers.

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


xdr_finalexample(xdrsp, finalp)
    XDR *xdrsp;
    struct finalexample *finalp;
{
    int i;

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

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

4.3 Lower Layers of RPC

In the examples given so far, RPC automatically takes care of many details for you. In this section, you'll see how to change the defaults by using the lower layers of the RPC library. This section assumes that you are familiar with sockets and the system calls for dealing with them.

There are several occasions when you may need to use lower layers of RPC. First, you may need to use TCP, since the higher layer uses UDP, which restricts RPC calls to 8K bytes of data. Using TCP permits calls to send long streams of data (see the TCP example later in this chapter).

Second, you may want to allocate and free memory while serializing or deserializing with XDR routines. There is no call at the higher level to let you free memory explicitly. For more explanation, see the "Memory Allocation with XDR" section below.

Third, you may need to perform authentication on either the client or server side, by supplying credentials or verifying them. See the explanation in the "Authentication" section that follows.

4.3.1 More Information About the Server

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

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

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

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

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

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

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

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

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

After creating an SVCXPRT, the next step is to call *pmap_unset()* so that if the *nusers* server crashed earlier, any previous trace of it is erased before restarting. More precisely, *pmap_unset()* erases the entry for RUSERS from the port mapper's tables.

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

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

The user service routine serializes the results and returns them to the RPC caller via *svc_sendreply()*. Its first parameter is the SVCXPRT handle, the

second is the XDR routine, and the third is a pointer to the data to be returned. Not illustrated previously is how a server handles an RPC program that passes data. As an example, we can add a procedure RUSERSPROC_BOOL, which has an argument *nusers*, and returns TRUE or FALSE depending on whether there are nusers logged on. It would look like this:

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

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

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

4.3.2 Memory Allocation with XDR

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

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

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

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

```
char chararr[SIZE];

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

where *chararr* has already allocated space. If you want XDR to do the allocation, you would have to rewrite this routine in the following way:

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

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

Then the RPC call might look like this:

```
char *arrptr;

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

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

To summarize, each XDR routine is responsible for serializing, deserializing, and freeing memory. When an XDR routine is called from *callrpc()*, the serializing part is used. When called from *svc_getargs()*, the deserializer is used. And when called from *svc_freeargs()*, the memory deallocator is used. When building simple examples like those in this section, a user doesn't have to worry about the three modes. The XDR chapter in this guide has examples of more sophisticated XDR routines that determine which of the three modes they are in to function correctly.

4.3.3 The Calling Side

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

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

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

    if (argc != 2) {
        fprintf(stderr, "usage: nusers hostname\n");
        exit(-1);
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
        herror(argv[1]);
        exit(-1);
    }
    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
```

```
                hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clntudp_create(&server_addr, RUSERSPROG,
          RUSERSVERS, pertry_timeout, &sock)) == NULL) {
       clnt_pcreateerror("clntudp_create");
       exit(-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void, 0,
          xdr_u_long, &nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(-1);
    }
    clnt_destroy(client);
    close(sock);
    exit(0);
}
```

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

The CLIENT pointer is encoded with the transport mechanism. *callrpc()* uses UDP, thus it calls *clntudp_create()* to get a CLIENT pointer. To specify TCP/IP, use *clnttcp_create()*.

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

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

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

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

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

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

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

4.4 Other RPC Features

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

4.4.1 Select on the Server Side

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

```
void
svc_run()
{
    fd_set readfds;
    int dtbsz = getdtablesize();

    for (;;) {
        readfds = svc_fdset;
        switch (select(dtbsz, &readfds, NULL,NULL,NULL)) {
            case -1:
                if (errno == EINTR)
                    continue;
                perror("select");
```

```
                return;
            case 0:
                break;
            default:
                svc_getreqset(&readfds);
        }
    }
}
```

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

4.4.2 Broadcast RPC

The *portmapper* is a daemon that converts RPC program numbers into UDP or TCP port numbers; see the *portmap*(1M) man page. You can't do broadcast RPC without the portmapper. Here are the main differences between broadcast RPC and normal RPC calls:

1. Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
2. Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UPD/IP.
3. The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
4. All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.
5. Broadcast requests are limited in size to the MTU (Maximum Transfer Unit) of the local network. For Ethernet, the MTU is 1500 bytes.

Broadcast RPC Synopsis

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

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

clnt_stat = clnt_broadcast_exp(prognum, versnum, procnum,
  inproc, in, outproc, out, eachresult, inittime, waittime)
    int        inittime;         /* initial wait period */
    int        waittime;         /* total wait period */
```

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

```
bool_t done;

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

If *done* is TRUE, then broadcasting stops and *clnt_broadcast()* returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with RPC_TIMEDOUT. Use *clnt_broadcast_exp()* to control the initial and total waiting intervals. To interpret *clnt_stat* errors, feed the error code to *clnt_perrno()*.

4.4.3 Batching

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

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

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

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

A contrived example of batching follows. Assume a string rendering service (like a window system) has two similar calls: one renders a string and returns void results, while the other renders a string and remains silent. The service (using the TCP/IP transport) may look like the following example.

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

void windowdispatch();

main()
{
    SVCXPRT *transp;

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

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

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        return;
    case RENDERSTRING:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can't decode arguments\n");
            /* tell caller he screwed up */
            svcerr_decode(transp);
            break;
        }
```

```
        /*
         * Code here to render the string s
         */
        if (!svc_sendreply(transp, xdr_void, NULL)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        break;

    case RENDERSTRING_BATCHED:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can't decode arguments\n");
            /*
             * We are silent in the face of protocol errors
             */
            break;
        }
        /*
         * Code here to render string s, but send no reply!
         */
        break;

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

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

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

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

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

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

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

    /* Now flush the pipeline */

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

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

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

4.4.4 Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network filesystem, require stronger security than what has been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type used as a default is type *none*.

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

The Client Side

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

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

the appropriate transport instance defaults the associate authentication handle to be

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

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

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

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

```
/*
 * Unix-style credentials.
 */
struct authunix_parms {
    /* credentials creation time */
    u_long      aup_time;

    /* host name of where the client is calling */
    char*aup_machname;

    /* client's UNIX effective uid */
    int aup_uid;

    /* client's current UNIX group id */
    int aup_gid;

    /* the element length of aup_gids array */
    u_int       aup_len;

    /* array of groups to which user belongs */
    int *aup_gids;
};
```

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

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

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

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

The Server Side

Service implementors have a harder time dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication style associated with it. Consider the fields of a request handle passed to a service dispatch routine.

```
/*
 * An RPC Service request
 */
struct svc_req {
    /* service program number */
    u_long  rq_prog;

    /* service protocol version number*/
    u_long  rq_vers;

    /* the desired procedure number*/
    u_long  rq_proc;

    /* raw credentials from the "wire" */
    struct opaque_auth rq_cred;

    /* read only, cooked credentials */
    caddr_t  rq_clntcred;
};
```

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

```
/*
 * Authentication info.  Mostly opaque to the programmer.
 */
struct opaque_auth {
    /* style of credentials */
    enum_t       oa_flavor;

    /* address of more auth stuff */
    caddr_t      oa_base;

    /* not to exceed MAX_AUTH_BYTES */
    u_int        oa_length;
};
```

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

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

2. That the request's *rq_clntcred* field is either NULL or points to a well formed structure that corresponds to a supported style of authentication credentials. Only UNIX style is currently supported, so (currently) *rq_clntcred* could be cast to a pointer to an *authunix_parms* structure. If *rq_clntcred* is NULL, the service implementor may wish to inspect the

other (opaque) fileds of *rq_cred* in case the service knows about a new type of authentication about which the RPC package does not know.

Our remote user's service example can be extended so that it computes results for all users except UID 16.

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

    /*
    * we don't care about authentication for
    * the null procedure
    */
    if (rqstp->rq_proc == NULLPROC) {
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        return;
    }
    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_UNIX:
        unix_cred =
            (struct authunix_parms *) rqstp->rq_clntcred;
        uid = unix_cred->aup_uid;
        break;
    case AUTH_NULL:
    default:
        svcerr_weakauth(transp);
        return;
    }
```

```
    switch (rqstp->rq_proc) {
    case RUSERSPROC_NUM:
        /*
         * make sure the caller is allow
         * to call this procedure.
         */
        if (uid == 16) {
            svcerr_systemerr(transp);
            return;
        }
        /*
         * code here to compute the number of
         * users and put in variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
    return;

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

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

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

4.4.5 Using Inetd

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

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

since *inet* passes a socket as file descriptor 0. Also, you should call *svc_register()* as:

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

with the final flag as 0, since the program will already be registered by *inetd*. Remember that if you want to exit from the server process and return control to *inetd*, you need to explicitly exit, since *svc_run()* never returns.

The format of entries in */usr/etc/inetd.conf* for RPC services is in one of the following two forms:

```
p_name/version dgram  rpc/udp wait user server args

p_name/version stream rpc/tcp wait user server args
```

where *p_name* is the symbolic name of the program as it appears in *rpc*(4), *server* is the program implementing the server, and *program* and *version* are the program and version numbers of the service. For more information, see the section on *inetd* in Chapter 3 and *inetd*(1M).

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

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

For server programs that handle multiple services or protocols, *inetd* allocates socket descriptors to protocols based on lexicographic order of service and protocol names.

4.5 More Examples

The following examples show a program version number, TCP, and callback procedure.

4.5.1 Versions Example

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

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

Both versions can be handled by the same C procedure:

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

    switch (rqstp->rq_proc) {
    case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        return;


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

4.5.2 TCP Example

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

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

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

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

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

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

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


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

    if ((hp = gethostbyname(host)) == NULL) {
        herror(host);
        return (-1);
    }
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
        hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clnttcp_create(&server_addr, prognum,
            versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
        perror("rpctcp_create");
```

```
        return (-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum,
            inproc, in, outproc, out, total_timeout);
    clnt_destroy(client);
    return (int)clnt_stat;
}


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

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

    if ((transp = svctcp_create(RPC_ANYSOCK,
            1024, 1024)) == NULL) {
        fprintf("svctcp_create: error\n");
        exit(1);
    }
    pmap_unset(RCPPROG, RCPVERS);
    if (!svc_register(transp, RCPPROG, RCPVERS, rcp_service,
            IPPROTO_TCP)) {
        fprintf(stderr, "svc_register: error\n");
        exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "svc_run should never return\n");
}
```

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

4.5.3 Callback Procedures

Occasionally, it is useful to have a server become a client, and make an RPC call back to the process that is its client, e.g., in remote debugging. The client is a window system program, and the server is a debugger running on a remote machine. Most of the time, the user clicks a mouse button at the debugging window that converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a breakpoint, the roles are reversed; the debugger wants to make an RPC call to the window program to tell the user that a breakpoint has been reached.

To do an RPC callback, you need a program number on which to make the RPC call. Since this will be a dynamically generated program number, it should be in the transient range, 0x40000000 – 0x5FFFFFFF. The routine *gettransient()* returns a valid program number in the transient range, and registers it with the portmapper. It only talks to the portmapper running on the same machine as the *gettransient()* routine itself. The call to *pmap_set()*

is a test and set operation, in that it indivisibly tests whether a program number has already been registered, and if it has not, then reserves it. On return, the *sockp* argument will contain a socket that can be used as the argument to an *svcudp_create()* or *svctcp_create()* call.

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

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

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

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

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

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

int callback();
char hostname[256];

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

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

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

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

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

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

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

docallback()
{
    int ans;

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

4.6 Synopsis of RPC and XDR Routines

auth_destroy()

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

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

authnone_create()

```
AUTH *
authnone_create()
```

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

authunix_create()

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

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

authunix_create_default()

```
AUTH *
authunix_create_default()
```

Calls *authunix_create()* with the appropriate parameters.

callrpc()

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

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

clnt_broadcast()

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

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

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

where *out* is the same as *out* passed to *clnt_broadcast()*, except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If *eachresult()* returns zero, *clnt_broadcast()* waits for more replies; otherwise it returns with appropriate status. Initially waits 4 seconds for a response before retrying. The next wait interval is doubled until it reaches a total wait time of 45 seconds. See also *clnt_setbroadcastbackoff()*.

clnt_broadcast_exp()

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

Like *clnt_broadcast()*, except you can specify the initial and total wait time. See also *clnt_setbroadcastbackoff()*.

clnt_call()

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

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

clnt_destroy()

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

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

clnt_freeres()

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

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

clnt_geterr()

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

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

clnt_pcreateerror()

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

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

clnt_perrno()

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

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

clnt_perror()

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

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

clnt_setbroadcastbackoff()

```
void
clnt_setbroadcastbackoff(first, next)
    void (*first)(struct timeval *tv);
    int (*next)(struct timeval *tv);
```

Set the timeout backoff iterator for clnt_broadcast(). The initial timeout is stored in *tv by first(). Subsequent timeouts are computed in *tv by next(), which returns 1 until a backoff limit is reached, and thereafter returns 0.

clnt_spcreateerror()

```
char *
clnt_spcreateerror(s)
    char *s;
```

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

clnt_sperrno()

```
char *
clnt_sperrno(stat)
    enum clnt_stat;
```

Returns a string corresponding to the condition indicated by *stat.*

clnt_sperror()

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

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

clnt_syslog()

```
void
clnt_syslog(clnt, s)
    CLIENT *clnt;
    char *s;
```

Logs an error to *syslog*(3) indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string `s` and a colon.

clntraw_create()

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

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

clnttcp_create()

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

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address **addr*. If *addr->sin_port* is zero, then it is set to the actual port that the remote program is listening on (the remote *portmap* service is consulted for this information). The parameter **sockp* is a socket; if it is RPC_ANYSOCK, then this routine opens a new one and sets **sockp* .

Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the parameters *sendsz* and *recvsz* ; values of zero choose suitable defaults. This routine returns NULL if it fails.

clntudp_create()

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

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

get_myaddress()

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

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

pmap_getmaps()

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

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

pmap_getport()

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

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

pmap_rmtcall()

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

A user interface to the *portmap* service, which instructs *portmap* on the host at IP address **addr* to make an RPC call on your behalf to a procedure on that host. The parameter **portp* will be modified to the program's port

number if the procedure succeeds. The definitions of other parameters are discussed in *callrpc()* and *clnt_call()*. See also *clnt_broadcast()*, *pmap_settimeouts()*, and *pmap_setrmtcalltimeout()*.

pmap_set()

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

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

pmap_setrmtcalltimeout()

```
void
pmap_setrmtcalltimeout(intertry)
    struct timeval intertry;
```

Set the retry timeout for *pmap_rmtcall()*. Note that the total timeout per call is an argument to *pmap_rmtcall()*.

pmap_settimeouts()

```
void
pmap_settimeouts(intertry, percall)
    struct timeval intertry, percall;
```

Set the retry and total timeouts for RPCs to the portmapper. These timeouts are used explicitly by *pmap_set()* and *pmap_getport()*, and implicitly by *clnt*_create()*.

pmap_unset()

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

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

registerrpc()

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

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

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

rpc_createerr

```
struct rpc_createerr    rpc_createerr;
```

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

_rpc_errorhandler()

```
void
_rpc_errorhandler(priority, format, ...);
    int priority;
    char *format;
```

Called by the RPC library routines to print an error message to *stderr* or to *syslog*(3), if *openlog*(3) was called. *priority* values are defined in <*syslog.h*>. *format* is printf-like format string. See comments in . <*rpc/errorhandler.h*> for details on defining your own version for more sophisticated error handling.

svc_destroy()

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

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

svc_fdset

```
fd_set  svc_fdset;
```

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

svc_freeargs()

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

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

svc_getargs()

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

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

svc_getcaller()

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

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

svc_getreq()

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

This routine is provided for compatibility with old code. Use svc_getreqset when developing new code.

svc_getreqset()

```
svc_getreqset(rdfds)
    fd_set *rdfds;
```

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

svc_register()

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

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

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

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

svc_run()

```
svc_run()
```

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

svc_sendreply()

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

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

svc_unregister()

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

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

svcerr_auth()

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

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

svcerr_decode()

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

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

svcerr_noproc()

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

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

svcerr_noprog()

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

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

svcerr_progvers()

```
void
svcerr_progvers(xprt, low_vers, high_vers)
    SVCXPRT *xprt;
    u_long low_vers, high_vers;
```

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

svcerr_systemerr()

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

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

svcerr_weakauth()

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

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

svcraw_create()

```
SVCXPRT *
svcraw_create()
```

This routine creates a toy RPC service transport, to which it returns a pointer. The transport is really a buffer within the process's address space, so the corresponding RPC client should live in the same address space; see

clntraw_create(). This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

svctcp_create()

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

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

svcudp_create()

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

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

xdr_accepted_reply()

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

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

xdr_array()

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

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

xdr_authunix_parms()

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

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

xdr_bool()

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

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

xdr_bytes()

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

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

xdr_callhdr()

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

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

xdr_callmsg()

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

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

xdr_char()

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

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

xdr_destroy()

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

A macro that invokes the destroy routine associated with the XDR stream, *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking *xdr_destroy*() is undefined.

xdr_double()

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

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

xdr_enum()

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

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

xdr_float()

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

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

xdr_getpos()

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

A macro that invokes the get-position routine associated with the XDR stream, *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

xdr_inline()

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

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

xdr_int()

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

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

xdr_long()

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

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

xdr_opaque()

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

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

xdr_opaque_auth()

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

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

xdr_pmap()

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

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

xdr_pmaplist()

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

Used for describing a list of port mappings, externally. This routine is useful for generating these parameters without using the *pmap* interface.

xdr_pointer()

```
xdr_pointer(xdrs, objpp, objsize, xdrobj)
    XDR *xdrs;
    char **objpp;
    u_int objsize;
    xdrproc_t xdrobj;
```

Like *xdr_reference( )* execpt that it serializes NULL pointers, whereas

xdr_reference( ) does not. Thus, *xdr_pointer( )* can represent recursive data structures, such as binary trees or linked lists.

xdr_reference()

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

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

xdr_rejected_reply()

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

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

xdr_replymsg()

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

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

xdr_setpos()

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

A macro that invokes the set position routine associated with the XDR stream *xdrs*. The parameter *pos* is a position value obtained from *xdr_getpos*(). This routine returns one if the XDR stream could be repositioned, and zero otherwise. Warning: it is difficult to reposition some types of XDR streams, so this routine may fail with one type of stream and succeed with another.

xdr_short()

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

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

xdr_string()

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

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

xdr_u_char()

```
xdr_u_char(xdrs, cp)
    XDR *xdrs;
    unsigned char *cp;
```

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

xdr_u_int()

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

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

xdr_u_long()

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

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

xdr_u_short()

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

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

xdr_union()

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

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

xdr_void()

```
xdr_void()
```

This routine always returns one.

xdr_wrapstring()

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

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

xdrmem_create()

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

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no more than *size* bytes long. The *op* determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

xdrrec_create()

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

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsize*; a value of zero indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsize*; it too can be set to a suitable default by passing a zero value. When a stream's output buffer is full, *writeit()* is called. Similarly, when a stream's input buffer is empty, *readit()* is called. The behavior of these two routines is similar to the UNIX system calls *read* and *write*, except that *handle* is passed to the former routines as the first parameter. Note that the XDR stream's *op* field must be set by the caller. Warning: this XDR stream implements an intermediate record stream. Therefore there are additional bytes in the stream to provide record boundary information.

xdrrec_endofrecord()

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

This routine can be invoked only on streams created by *xdrrec_create()*. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is non-zero. This routine returns one if it succeeds, zero otherwise.

xdrrec_eof()

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

This routine can be invoked only on streams created by *xdrrec_create()*. After consuming the rest of the current record in the stream, this routine returns one if the stream has no more input, zero otherwise.

xdrrec_skiprecord()

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

This routine can be invoked only on streams created by *xdrrec_create*(). It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns one if it succeeds, zero otherwise.

xdrstdio_create()

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

This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the Standard I/O stream *file*. The parameter *op* determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE). Warning: the destroy routine associated with such XDR streams calls *fflush*() on the *file* stream, but never *close*().

xprt_register()

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

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

xprt_unregister()

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

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

5. The *rpcgen* Compiler

This chapter covers the following topics:

- converting local procedures into remote procedures
- generating XDR routines
- the C-preprocessor
- the rpcgen programming guide
- the RPC language

The details of programming applications to use Remote Procedure Calls can be overwhelming. Perhaps most daunting is the writing of the XDR routines necessary to convert procedure arguments and results into their network format and vice-versa.

Fortunately, *rpcgen*(1) exists to help you write RPC applications simply and directly. *rpcgen* does most of the dirty work; you just debug the main features of the application instead of spending most of your time debugging network interface code.

rpcgen is a compiler. It accepts a remote program interface definition written in a language, called RPC Language, which is similar to C. It produces a C language output that includes stub versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, and a header file that contains common definitions. The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures that are to be invoked by remote clients.

You can compile and link *rpcgen*'s output files in the usual way. The developer writes server procedures—in any language that observes calling conventions—and links them with the server skeleton produced by *rpcgen* to get an executable server program. To use a remote program, a

programmer writes an ordinary main program that makes local procedure calls to the client stubs produced by *rpcgen*. Linking this program with *rpcgen*'s stubs creates an executable program. (At present the main program must be written in C.)

You can use *rpcgen* options to suppress stub generation, to specify the transport to be used by the server stub, and to pass flags to *cpp* or choose a different preprocessor.

Like all compilers, *rpcgen* reduces development time that would otherwise be spent coding and debugging low-level routines. All compilers, including *rpcgen*, do this at a small cost in efficiency and flexibility. However, many compilers allow escape hatches for programmers to mix low-level code with high-level code. *rpcgen* is no exception. In speed-critical applications, hand-written routines can be linked with the *rpcgen* output without any difficulty. Also, you may proceed by using *rpcgen* output as a starting point, and then rewriting it as necessary. (If you need a discussion of RPC programming without *rpcgen*, see the previous chapter.)

5.1 Converting Local Procedures into Remote Procedures

Assume an application that runs on a single machine, one that you want to convert to run over the network. Following is an example of such a conversion—a program that prints a message to the console.

```
/*
 * printmsg.c: print a message on the console
 */
#include <stdio.h>

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

    if (argc < 2) {
        fprintf(stderr, "usage: %s <message>\n", argv[0]);
        exit(1);
    }
    message = argv[1];

    if (!printmessage(message)) {
        fprintf(stderr, "%s: couldn't print your message\n",
                argv[0]);
        exit(1);
    }
    printf("Message Delivered!\n");
    exit(0);
}
/*
 * Print a message to the console.  Return a boolean
 * indicating whether the message was actually printed.
 */
printmessage(msg)
    char *msg;
{
    FILE *f;

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

And then, of course:

```
% cc printmsg.c  -o printmsg
% printmsg "Hello, there."
Message Delivered!
%
```

If *printmessage()* was turned into a remote procedure, then it could be called from anywhere in the network. Ideally, you would like to insert a keyword such as *remote* in front of a procedure to turn it into a remote procedure. Unfortunately, you have to live within the constraints of the C language, since it existed long before RPC did. But even without language support, it's not very difficult to make a procedure remote.

In general, it's necessary to figure out what the types are for all procedure inputs and outputs. In this case, there is a procedure, *printmessage()*, that takes a string as input, and returns an integer as output. Knowing this, you can write a protocol specification in RPC language that describes the remote version of *printmessage()*:

```
/*
 * msg.x: Remote message printing protocol
 */

program MESSAGEPROG {
        version MESSAGEVERS {
                int PRINTMESSAGE(string) = 1;
        } = 1;
} = 99;
```

Remote procedures are part of remote programs, so an entire remote program was declared here that contains the single procedure *PRINTMESSAGE*. This procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary because *rpcgen* generates it automatically.

Notice that everything is declared with all capital letters. This is not required, but is a good convention to follow.

Notice also that the argument type is `string` and not `char *`. This is because a `char *` in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a `string`.

There are just two more things to write. First, there is the remote procedure itself. The following example defines a remote procedure to implement the *PRINTMESSAGE* procedure declared above.

```
/*
 * msg_proc.c:  implementation of the remote
 *              procedure "printmessage"
 */

#include <stdio.h>
#include <rpc/rpc.h>      /* always needed  */
#include "msg.h"          /* need this too: msg.h will be generated by rpcgen */

/*
 * Remote version of "printmessage"
 */

int *
printmessage_1(msg)
        char **msg;
{
    static int result;   /* must be static! */
    FILE *f;

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

Notice here that the declaration of the remote procedure *printmessage_1()* differs from that of the local procedure *printmessage()* in three ways:

1. It takes a pointer to a string instead of a string itself. This is true of all remote procedures: they always take pointers to their arguments rather than the arguments themselves.

2. It returns a pointer to an integer instead of an integer itself. This is also generally true of remote procedures: they always return a pointer to their results.

3. It has an **_1** appended to its name. In general, all remote procedures called by *rpcgen* are named by the following rule: the name in the program definition (here *PRINTMESSAGE*) is converted to all lower-case letters, an underbar (_) is appended to it, and finally the version number (here 1) is appended.

Finally, declare the main client program that will call the remote procedure:

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include <rpc/rpc.h>        /* always needed */
#include "msg.h"            /* need this too: msg.h will be generated by rpcgen */

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

    if (argc < 3) {
         fprintf(stderr, "usage: %s host message\n", argv[0]);
         exit(1);
    }
     /* Save values of command line arguments */
    server = argv[1];
    message = argv[2];
    /*
     * Create client "handle" used for calling
     * MESSAGEPROG on the server designated on the
     * command line. We tell the RPC package to use the
     * "tcp" protocol when contacting the server.
     */
    cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
    if (cl == NULL) {
            /*
             * Couldn't establish connection with server.
             * Print error message and die.
             */
            clnt_pcreateerror(server);
            exit(1);
    }
    /* Call the remote procedure "printmessage" on the server /*
    result = printmessage_1(&message, cl);
    if (result == NULL) {
```

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

There are two things to note:

1. A client *handle* is created using the RPC library routine *clnt_create()*. This client handle will be passed to the stub routines that call the remote procedure.

2. The remote procedure *printmessage_1()* is called exactly the same way as it is declared in *msg_proc.c* except for the inserted client handle as the first argument.

The following example shows how to put all of the pieces together.

```
% rpcgen msg.x
% cc rprintmsg.c msg_clnt.c -lsun -o rprintmsg
% cc msg_proc.c msg_svc.c -lsun -o msg_server
```

Two programs were compiled: the client program *rprintmsg* and the server program *msg_server*. Before compilation, *rpcgen* was used to fill in the missing pieces. The list that follows explains what *rpcgen* did with the input file *msg.x*.

1. It created a header file called *msg.h* that contained *#define*'s for *MESSAGEPROG*, *MESSAGEVERS* and *PRINTMESSAGE* for use in the other modules.

2. It created client *stub* routines in the *msg_clnt.c* file. In this case there is only one, the *printmessage_1()* that was referred to from the *printmsg* client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is *FOO.x*, the client stubs output file is called *FOO_clnt.c*.

3. It created the server program that calls *printmessage_1()* in *msg_proc.c*. This server program is named *msg_svc.c*. The rule for naming the server output file is similar to the previous one: for an input file called *FOO.x*, the output server file is named *FOO_svc.c*.

Now you're ready to have some fun. First, copy the server to a remote machine and run it. For this example, the machine is called *clyde*. Server processes are run in the background, because they never exit.

```
clyde% msg_server &
```

Then on your local machine (*bonnie*), print a message on clyde's console.

```
bonnie% printmsg clyde "Hello, clyde."
```

The message will get printed to clyde's console. You can print a message on anybody's console (including your own) with this program if you are able to copy the server to their machine and run it.

5.2 Generating XDR Routines

The previous example only demonstrated the automatic generation of client and server RPC code. *rpcgen* may also be used to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice-versa. This example presents a complete RPC service—a remote directory listing service, which uses *rpcgen* not only to generate stub routines, but also to generate the XDR routines.

The protocol description file appears in the next example.

```
 /* dir.x: Remote directory listing protocol */

const MAXNAMELEN = 255;          /* maximum length of a directory entry */

typedef string nametype<MAXNAMELEN>;     /* a directory entry */

typedef struct namenode *namelist;       /* a link in the listing */

 /* A node in the directory listing */

struct namenode {
        nametype name;           /* name of directory entry */
        namelist next;           /* next entry */
};

 /* The result of a READDIR operation. */

union readdir_res switch (int errno) {
case 0:
        namelist list;   /* no error: return directory listing */
default:
        void;            /* error occurred: nothing else to return */
};

/*
 * The directory program definition
 */
program DIRPROG {
        version DIRVERS {
                readdir_res
                READDIR(nametype) = 1;
        } = 1;
} = 76;
```

Note: Define types (like *readdir_res* in the example above) by using the `struct`, `union` and `enum` keywords; those keywords should not be used in subsequent declarations of variables of those types. For example, if you define a union `foo`, you should declare using only `foo` and not `union foo`. In fact, *rpcgen* compiles RPC unions into C structures; it is an error to declare them using the `union` keyword.

Running *rpcgen* on *dir.x* creates four output files. Three are the same as before: header file, client stub routines and server skeleton. The fourth are the XDR routines necessary for converting the data types you declared into XDR format and vice-versa. These are output in the file *dir_xdr.c*.

The *READDIR* procedure is implemented as shown in the following example.

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>
#include <sys/dir.h>
#include "dir.h"

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

readdir_res *
readdir_1(dirname)
        nametype *dirname;
{
    DIR *dirp;
    struct direct *d;
    namelist nl;
    namelist *nlp;
    static readdir_res res; /* must be static! */

    /*
     * Open directory
     */
    dirp = opendir(*dirname);
    if (dirp == NULL) {
            res.errno = errno;
            return (&res);
    }
```

```
    /*
     * Free previous result
     */
    xdr_free(xdr_readdir_res, &res);

    /*
     * Collect directory entries.
     * Memory allocated here will be freed by xdr_free
     * next time readdir_1 is called
     */
    nlp = &res.readdir_res_u.list;
    while (d = readdir(dirp)) {
            nl = *nlp = (namenode *) malloc(sizeof(namenode));
            nl->name = strdup(d->d_name);
            nlp = &nl->next;
    }
    *nlp = NULL;

    /*
     * Return the result
     */
    res.errno = 0;
    closedir(dirp);
    return (&res);
}
```

Finally, there is the client side program to call the server:

```
/*
 * rls.c: Remote directory listing client
 */
#include <stdio.h>
#include <rpc/rpc.h>      /* always need this */
#include "dir.h" /* will be generated by rpcgen */

extern int errno;

main(argc, argv)
        int argc;
        char *argv[];
{
    CLIENT *cl;
    char *server;
    char *dir;
    readdir_res *result;
    namelist nl;
```

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

/*
 * Remember what our command line arguments refer to
 */
server = argv[1];
dir = argv[2];

/*
 * Create client "handle" used for calling
 * MESSAGEPROG on the server designated on the
 * command line. We tell the RPC package to use the
 * "tcp" protocol when contacting the server.
 */
cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
if (cl == NULL) {
        /*
         * Couldn't establish connection with server.
         * Print error message and die.
         */
        clnt_pcreateerror(server);
        exit(1);
}

/*
 * Call the remote procedure readdir on the server
 */
result = readdir_1(&dir, cl);
if (result == NULL) {
        /*
         * An error occurred while calling the server.
         * Print error message and die.
         */
        clnt_perror(cl, server);
        exit(1);
}

/*
 * Okay, the remote procedure was called successfully.
 */
if (result->errno != 0) {
        /*
         * A remote system error occurred.
         * Print error message and die.
         */
        errno = result->errno;
        perror(dir);
        exit(1);
}
```

```
    /*
     * Successfully got a directory listing.
     * Print it out.
     */
    for (nl = result->readdir_res_u.list; nl != NULL;
      nl = nl->next) {
            printf("%s\n", nl->name);
    }
    exit(0);
}
```

Compile everything, and run:

```
bonnie% rpcgen dir.x
bonnie% cc rls.c dir_clnt.c dir_xdr.c -lsun -o rls
bonnie% cc dir_svc.c dir_proc.c dir_xdr.c -lsun -o dir_svc

bonnie% dir_svc &

clyde% rls bonnie /usr/pub
.
..
apseqnchar
cateqnchar
eqnchar
psceqnchar
terminals
clyde%
```

A final note about *rpcgen*: You can test the client program and the server procedure together as a single program by simply linking them with each other rather than with the client and server stubs. The procedure calls will be executed as ordinary local procedure calls and the program can be debugged with a local debugger such as *dbx*. When the program is working, the client program can be linked to the client stub produced by *rpcgen* and the server procedures can be linked to the server stub produced by *rpcgen*.

Note that if you do this, you may want to comment out calls to RPC library routines, and have client-side routines call server routines directly.

5.3 The C-Preprocessor

The C-preprocessor is run on all input files before they are compiled, so all the preprocessor directives are legal within a **.x** file. Four symbols may be defined, depending on which output file is being generated. Symbols are:

Symbol	Usage
RPC_HDR	for header-file output
RPC_XDR	for XDR routine output
RPC_SVC	for server-skeleton output
RPC_CLNT	for client stub output

Also, *rpcgen* does a little preprocessing of its own. Any line that begins with a percent sign is passed directly into the output file, without any interpretation of the line. The following example demonstrates the preprocessing features.

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
        version TIMEVERS {
                unsigned int TIMEGET(void) = 1;
        } = 1;
} = 44;

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

The percent (%) feature is not generally recommended, as there is no guarantee that the compiler will stick the output where you intended.

5.4 *rpcgen* Programming Guide

This section describes timeout changes, broadcast on the server side, and information passed to server procedures.

5.4.1 Timeout Changes

RPC sets a default timeout of 25 seconds for RPC calls when *clnt_create()* is used. This timeout may be changed using *clnt_control()*. The following code fragment demonstrates use of *clnt_control ()*:

```
struct timeval tv;
CLIENT *cl;

cl = clnt_create("somehost", SOMEPROG, SOMEVERS, "tcp");
if (cl == NULL) {
        exit(1);
}
tv.tv_sec = 60;  /* change timeout to 1 minute */
tv.tv_usec = 0;
clnt_control(cl, CLSET_TIMEOUT, &tv);
```

5.4.2 Handling Broadcast on the Server Side

When a procedure is known to be called via broadcast RPC, it is usually wise for the server not to reply unless it can provide some useful information to the client. This prevents the network from getting flooded by useless replies.

To prevent the server from replying, a remote procedure can return NULL as its result, and the server code generated by *rpcgen* will detect this and not send out a reply.

The next example shows a procedure that replies only if it thinks it is an NFS server.

```
void *
reply_if_nfsserver()
{
        char notnull;     /* just here so you can use its address */
        if (access("/etc/exports", F_OK) < 0) {
                return (NULL);   /* prevent RPC from replying */
        }
        /*
         * return non-null pointer so RPC will send out a reply
         */
        return ((void *)&notnull);
}
```

Note that if procedure returns type `void *`, they must return a non-NULL pointer if they want RPC to reply for them.

5.4.3 Other Information Passed to Server Procedures

Server procedures will often want to know more about an RPC call than just its arguments. For example, getting authentication information is important to procedures that want to implement some level of security. This extra information is actually supplied to the server procedure as a second argument. The following example demonstrates its use. The previous *printmessage_1()* procedure has been rewritten to only allow root users to print a message to the console.

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

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

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

5.5 RPC Language

RPC language is an extension of XDR language. The sole extension is the addition of the *program* type. For a description of the XDR language syntax, see Chapter 6. For a description of the RPC extensions to the XDR language, see the Chapter 7.

XDR language is so close to C that if you know C, you know most of it already. This section describes the syntax of the RPC language, and explains how the various RPC and XDR type definitions get compiled into C-type definitions in the output header file.

5.5.1 Definitions

An RPC language file consists of a series of definitions:

```
definition-list:
    definition ";"
    definition ";" definition-list
```

It recognizes five types of definitions:

```
definition:
    enum-definition
    struct-definition
    union-definition
    typedef-definition
    const-definition
    program-definition
```

5.5.2 Structures

An XDR struct is declared almost exactly like its C counterpart.

```
struct-definition:
    "struct" struct-ident "{"
        declaration-list
    "}"

declaration-list:
    declaration ";"
    declaration ";" declaration-list
```

As an example, here is an XDR structure to a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file.

```
struct coord {              struct coord {
     int x;        -->           int x;
     int y;                      int y;
};                          };
typedef struct coord coord;
```

The output is identical to the input, except for the added *typedef* at the end of the output. This allows you to use `coord` instead of `struct coord` when declaring items.

5.5.3 Unions

XDR unions are discriminated unions, and look different from C unions. They are more analogous to Pascal variant records than they are to C unions.

```
union-definition:
    "union" union-ident "switch" "(" declaration ")" "{"
        case-list
    "}"
case-list:
    "case" value ":" declaration ";"
    "default" ":" declaration ";"
    "case" value ":" declaration ";" case-list
```

The next example shows a type that might be returned as the result of a **read data** operation. If no error, return a block of data. Otherwise, return nothing.

```
union read_result switch (int errno) {
case 0:
    opaque data[1024];
default:
    void;
};
```

It gets compiled into the following:

```
struct read_result {
    int errno;
    union {
        char data[1024];
    } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the name as the type name, except for the trailing **_u**.

5.5.4 Enumerations

XDR enumerations have the same syntax as C enumerations.

```
enum-definition:
    "enum" enum-ident "{"
        enum-value-list
    "}"

enum-value-list:
    enum-value
    enum-value "," enum-value-list

enum-value:
    enum-value-ident
    enum-value-ident "=" value
```

The XDR *enum* and the C *enum* get compiled into:

```
enum colortype {        enum colortype {
    RED = 0,                RED = 0,
    GREEN = 1,    -->       GREEN = 1,
    BLUE = 2                BLUE = 2,
};                      };
typedef enum colortype colortype;
```

5.5.5 Typedef

XDR typedefs have the same syntax as C typedefs.

```
typedef-definition:
    "typedef" declaration
```

The following example defines a *fname_type* used for declaring file name strings that have a maximum length of 255 characters.

```
typedef string fname_type<255>; --> typedef char *fname_type;
```

5.5.6 Constants

XDR constants symbolic constants that may be used wherever a integer constant is used, for example, in array size specifications.

```
const-definition:
    "const" const-ident "=" integer
```

For example, the following defines a constant *DOZEN* equal to 12.

```
const DOZEN = 12;  -->  #define DOZEN 12
```

5.5.7 Programs

RPC programs are declared using the following syntax:

```
program-definition:
    "program" program-ident "{"
        version-list
    "}" "=" value

version-list:
    version ";"
    version ";" version-list

version:
    "version" version-ident "{"
        procedure-list
    "}" "=" value
```

```
    procedure-list:
        procedure ";"
        procedure ";" procedure-list

    procedure:
        type-ident procedure-ident "(" type-ident ")" "=" value
```

The following example shows the time protocol, revisited.

```
/*
 * time.x: Get or set the time. Time is represented as number of
 * seconds since 0:00, January 1, 1970.
 */
program TIMEPROG {
    version TIMEVERS {
        unsigned int TIMEGET(void) = 1;
        void TIMESET(unsigned) = 2;
    } = 1;
} = 44;
```

This file compiles into #defines in the output header file:

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

5.5.8 Declarations

In XDR, there are only four kinds of declarations:

simple-declaration
fixed-array-declaration
variable-array-declaration
pointer-declaration

- **Simple declarations** are just like simple C declarations.

```
simple-declaration:
    type-ident variable-ident
```

Example:

```
colortype color;     --> colortype color;
```

- **Fixed-length Array Declarations** are just like C array declarations:

```
fixed-array-declaration:
    type-ident variable-ident "[" value "]"
```

Example:

```
colortype palette[8];     --> colortype palette[8];
```

- **Variable-Length Array Declarations** have no explicit syntax in C, so XDR invents its own using angle-brackets.

```
variable-array-declaration:
    type-ident variable-ident "<" value ">"
    type-ident variable-ident "<" ">"
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size.

```
int heights<12>;      /* at most 12 items */
int widths<>;         /* any number of items */
```

Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into `structs`. For example, the `heights` declaration gets compiled into the following struct:

```
struct {
    u_int heights_len;   /* # of items in array */
    int *heights_val;    /* pointer to array */
} heights;
```

The number of items in the array is stored in the **_len** component and the pointer to the array is stored in the **_val** component. The first part of each component's name is the same as the name of the declared XDR variable.

- **Pointer Declarations** are made in XDR exactly as they are in C. You can't really send pointers over the network, but you can use XDR pointers for sending recursive data types such as lists and trees. The type is actually called *optional-data*, not *pointer*, in XDR language.

```
pointer-declaration:
    type-ident "*" variable-ident
```

Example:

```
listitem *next;  -->  listitem *next;
```

5.5.9 Special Cases

There are a few exceptions to the rules described above.

Booleans: C has no built-in boolean type. However, the RPC library does have a boolean type called *bool_t* that is either *TRUE* or *FALSE*. Things declared as type *bool* in XDR language are compiled into *bool_t* in the output header file. For example:

```
bool married;  -->  bool_t married;
```

Strings: C has no built-in string type, but instead uses the null-terminated `char *` convention. In XDR language, strings are declared using the `string` keyword, and compiled into `char *`s in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the *NULL* character). The maximum size may be left off, indicating a string of arbitrary length. Two examples are:

```
string name<32>;     -->  char *name;
string longname<>;   -->  char *longname;
```

Opaque Data: Opaque data is used in RPC and XDR to describe untyped data, that is, just sequences of arbitrary bytes. It may be declared either as a fixed or variable length array. Examples are:

```
opaque diskblock[512];  -->  char diskblock[512];

opaque filedata<1024>;  -->  struct {
                                u_int filedata_len;
                                char *filedata_val;
                             } filedata;
```

Voids: In a void declaration, the variable is not named. The declaration is just `void` and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure).

6. XDR Programming

This chapter contains technical notes on Sun's implementation of the External Data Representation (XDR) standard, a set of library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. The chapter concludes with the formal specification of the XDR standard. XDR is the backbone of Sun's Remote Procedure Call package, in the sense that data for remote procedure calls is transmitted using the standard. XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine.

This chapter contains:

- a short tutorial overview of the XDR library routines
- a guide to accessing currently available XDR streams
- information on defining new streams and data types
- the XDR protocol specification.

XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) will only need the information in the "Number Filters," "Floating Point Filters," and "Enumeration Filters" sections.

Note: You can use *rpcgen* to write XDR routines even in cases where no RPC calls are being made.

On IRIX, C programs that want to use XDR routines must include the file *<rpc/rpc.h>*, which contains all the necessary interfaces to the XDR system. Since the C library *libsun.a* contains all the XDR routines, compile using:

```
cc prog.c -lsun -o prog
```

See compiling BSD programs in Chapter 3 for other compiling hints.

Justification

Consider two programs, *writer* and *reader*. *writer* looks like this:

```
#include <stdio.h>

main()              /* writer.c */
{
    long i;

    for (i = 0; i < 8; i++) {
        if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
    }
    exit(0);
}
```

and *reader* looks like this:

```
#include <stdio.h>

main()              /* reader.c */
{
    long i, j;

    for (j = 0; j < 8; j++) {
        if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

The two programs appear to be portable, because they pass *lint* checking, and they exhibit the same behavior when executed on different hardware architectures, an IRIS-4D and a VAX. Piping the output of the *writer* program to the *reader* program produces identical results on both machines.

```
iris% writer | reader
0 1 2 3 4 5 6 7

vax% writer | reader
0 1 2 3 4 5 6 7
```

With the advent of local area networks and Berkeley's 4.2 BSD UNIX came the concept of "network pipes" — a process produces data on one machine, and a second process consumes data on another machine. A network pipe can be constructed with *writer* and *reader*. The following results show if the first produces data on an IRIS, and the second consumes data on a VAX.

```
iris% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296 117440512
iris%
```

Identical results can be obtained by executing *writer* on the VAX and *reader* on the IRIS-4D. These results occur because the byte ordering of long integers differs between the VAX and the IRIS, even though word size is the same. Note that 16777216 is 2^{24} — when four bytes are reversed, the 1 winds up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the *read()* and *write()* calls with calls to an XDR library routine *xdr_long()*, a filter that knows the standard representation of a long integer in its external form. The following example shows the revised versions of *writer*.

```
#include <stdio.h>
#include <rpc/rpc.h>        /* xdr is a sub-library of rpc */

main()                      /* writer.c */
{
    XDR xdrs;
    long i;
```

```
    xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
    for (i = 0; i < 8; i++) {
        if (! xdr_long(&xdrs, &i)) {
            fprintf(stderr, "failed!\n");
            exit(1);
        }
        exit(0);
}
```

and *reader*:

```
#include <stdio.h>
#include <rpc/rpc.h>        /* xdr is a sub-library of rpc */
main()                      /* reader.c */
{
    XDR xdrs;
    long i, j;

    xdrstdio_create(&xdrs, stdin, XDR_DECODE);
    for (j = 0; j < 8; j++) {
        if (! xdr_long(&xdrs, &i)) {
           fprintf(stderr, "failed!\n");
           exit(1);
        }
        printf("%ld ", i);
    }
    printf("\n");
    exit(0);
}
```

The new programs were executed on an IRIS, on a VAX, and from an IRIS to a VAX; the results are shown below.

```
iris% writer | reader
0 1 2 3 4 5 6 7

vax% writer | reader
0 1 2 3 4 5 6 7

iris% writer | rsh vax reader
0 1 2 3 4 5 6 7
```

Dealing with integers is just the tip of the portable-data iceberg. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. Pointers are convenient to use, but have no meaning outside the machine where they are defined.

6.1 A Canonical Standard

XDR's approach to standardizing data representations is *canonical.* That is, XDR defines a single byte order ("big-endian"), a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations; similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents. The single standard completely decouples programs that create or send portable data from those that use or receive portable data.

The advent of a new machine or a new language has no effect upon the community of existing portable data creators and users. A new machine joins this community by being taught how to convert the standard representations and its local representations; the local representations of other machines are irrelevant. Conversely, to existing programs running on other machines, the local representations of the new machine are also irrelevant; such programs can immediately read portable data produced by the new machine because such data conforms to the canonical standard that they already understand.

There are strong precedents for XDR's canonical approach. For example, TCP/IP, UDP/IP, XNS, Ethernet, and, indeed, all protocols below layer five of the ISO model, are canonical protocols. The advantage of any canonical approach is simplicity; in the case of XDR, a single set of conversion routines is written once and is never touched again. The canonical approach has a disadvantage, but it is unimportant in real-world data transfer applications.

Suppose two little-endian machines are transferring integers according to the XDR standard. The sending machine converts the integers from little-endian byte order to XDR (big-endian) byte order; the receiving machine performs the reverse conversion. Because both machines observe the same byte order, their conversions are unnecessary. The point, however, is not necessity, but cost as compared to the alternative.

The time spent converting to and from a canonical representation is insignificant, especially in networking applications. Most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure. To transmit a tree, for example, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there; storage for the leaf may have to be deallocated as well. Similarly, to receive a tree, storage must be allocated for each leaf, data must be moved from the buffer to the leaf and properly aligned, and pointers must be constructed to link the leaves together. Every machine pays the cost of traversing and copying data structures whether or not conversion is required.

In networking applications, communications overhead—the time required to move the data down through the sender's protocol layers, across the network and up through the receiver's protocol layers—dwarfs conversion overhead.

6.2 The XDR Library

The XDR library not only solves data portability problems, it also allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it can make sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, you can write your own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements, or pointers to other structures.

Examine the two programs more closely. There is a family of XDR stream creation routines in which each member treats the stream of bits differently. In the example, data is manipulated using standard I/O routines; therefore,

use *xdrstdio_create*(). The parameters to XDR stream creation routines vary according to their function. In our example, *xdrstdio_create()* takes a pointer to an XDR structure that it initializes, a pointer to a *FILE* that the input or output is performed on, and the operation. The operation may be *XDR_ENCODE* for serializing in the *writer* program, or *XDR_DECODE* for deserializing in the *reader* program.

Note: RPC users never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the users.

The *xdr_long()* primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns FALSE (0) if it fails, and TRUE (1) if it succeeds. Second, for each data type, *xxx*, there is an associated XDR routine of the form shown in the following example.

```
xdr_xxx(xdrs, xp)
        XDR *xdrs;
        xxx *xp;
{
}
```

In our case, *xxx* is long, and the corresponding XDR routine is a primitive, *xdr_long()*. The client could also define an arbitrary structure *xxx* in which case the client would also supply the routine *xdr_xxx*(), describing each field by calling XDR routines of the appropriate type. In all cases the first parameter, *xdrs* can be treated as an opaque handle, and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The idea is to call the same routine for either operation — this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself — only in the case of deserialization is the object modified. This feature is not shown in our trivial example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, the user can obtain the direction of the XDR operation. See the "XDR Operation Directions" section below for details.

Look at a slightly more complicated example. Assume that a person's gross assets and liabilities are to be exchanged among processes. Also assume that these values are important enough to warrant their own data type.

```
struct gnumbers {
    long g_assets;
    long g_liabilities;
};
```

The corresponding XDR routine describing this structure would be:

```
bool_t          /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
     XDR *xdrs;
     struct gnumbers *gp;
{
     if (xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities))
           return(TRUE);
        return(FALSE);
}
```

Note that the parameter *xdrs* is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return FALSE if the subroutine fails.

This example also shows that the type *bool_t* is declared as an integer whose only values are *TRUE* (1) and *FALSE* (0). This document uses the following definitions:

```
#define bool_t   int
#define TRUE     1
#define FALSE    0
```

Keeping these conventions in mind, *xdr_gnumbers()* can be rewritten as follows:

```
xdr_gnumbers(xdrs, gp)
   XDR *xdrs;
   struct gnumbers *gp;
{
   return (xdr_long(xdrs, &gp->g_assets) &&
        xdr_long(xdrs, &gp->g_liabilities));
}
```

This document uses both coding styles.

6.3 XDR Library Primitives

This section gives a synopsis of each XDR primitive. It starts with basic data types and moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the include file *<rpc/xdr.h>*, automatically included by *<rpc/rpc.h>*.

6.3.1 Number Filters

The XDR library provides primitives to translate between numbers and their corresponding external representations. Primitives cover the set of numbers in:

```
[signed, unsigned] * [short, int, long]
```

Specifically, the eight primitives are:

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

bool_t xdr_u_char(xdrs, ucp)
        XDR *xdrs;
        unsigned char *ucp;

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

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

bool_t xdr_long(xdrs, lip)
        XDR *xdrs;
        long *lip;

bool_t xdr_u_long(xdrs, lup)
        XDR *xdrs;
        u_long *lup;
```

```
bool_t xdr_short(xdrs, sip)
        XDR *xdrs;
        short *sip;

bool_t xdr_u_short(xdrs, sup)
        XDR *xdrs;
        u_short *sup;
```

The first parameter, *xdrs*, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return TRUE if they complete successfully, and FALSE otherwise.

6.3.2 Floating Point Filters

The XDR library also provides primitive routines for C's floating point types:

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


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

The first parameter, *xdrs* is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. Both routines return *TRUE* if they complete successfully, and *FALSE* otherwise.

Note: Since the numbers are represented in IEEE floating point, routines may fail when decoding a valid IEEE representation into a machine-specific representation, or vice-versa.

6.3.3 Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C *enum* has the same representation inside the machine as a C integer. The boolean type is an important instance of the *enum*. The external representation of a boolean is always one (TRUE) or zero (FALSE).

```
#define bool_t   int
#define FALSE    0
#define TRUE     1

#define enum_t int

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

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

The second parameters *ep* and *bp* are addresses of the associated type that provides data to, or receives data from, the stream *xdrs*. The routines return TRUE if they complete successfully, and FALSE otherwise.

6.3.4 No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();  /* always returns TRUE */
```

6.3.5 Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions then the primitives discussed above. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with XDR_DECODE.

Therefore, the XDR package must provide means to deallocate memory. This is done by an XDR operation, XDR_FREE. To review, the three XDR directional operations are XDR_ENCODE, XDR_DECODE, and XDR_FREE.

Strings

In C, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a *char* * and not a sequence of characters. The external representation of a string is drastically different from its internal representation. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers. Conversion between the two representations is accomplished with the routine *xdr_string()*:

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

The first parameter *xdrs* is the XDR stream handle. The second parameter *sp* is a pointer to a string (type *char* **) . The third parameter *maxlength* specifies the maximum number of bytes allowed during encoding or decoding. its value is usually specified by a protocol. For example, a protocol specification may say that a file name may be no longer than 255 characters.

The routine returns *FALSE* if the number of characters excee *maxlength*, and *TRUE* if it doesn't.

Note: Keep *maxlength* small. If it is too big you can overrun the heap, since *xdr_string()* will call *malloc()* for space.

The behavior of *xdr_string()* is similar to the behavior of other routines discussed in this section. The direction XDR_ENCODE is easiest to understand. The parameter *sp* points to a string of a certain length; if the string does not exceed *maxlength*, the bytes are serialized.

The effect of deserializing a string is subtle. First the length of the incoming string is determined; it must not exceed *maxlength*. Next *sp* is dereferenced; if the the value is NULL, then a string of the appropriate length is allocated

and **sp* is set to this string. If the original value of **sp* is non-NULL, then the XDR package assumes that a target area has been allocated, which can hold strings no longer than *maxlength*. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the XDR_FREE operation, the string is obtained by dereferencing *sp*. If the string is not NULL, it is freed and **sp* is set to NULL. In this operation, *xdr_string()* ignores the *maxlength* parameter.

Byte Arrays

Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways: 1) the length of the array (the byte count) is explicitly located in an unsigned integer, 2) the byte sequence is not terminated by a null character, and 3) the external representation of the bytes is the same as their internal representation. The primitive *xdr_bytes()* converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
        XDR *xdrs;
        char **bpp;
        u_int *lp;
        u_int maxlength;
```

The usage of the first, second and fourth parameters are identical to the first, second and third parameters of *xdr_string*(), respectively. The length of the byte area is obtained by dereferencing *lp* when serializing; **lp* is set to the byte length when deserializing.

Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. *xdr_bytes()* treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is built-in. The generic array primitive, *xdr_array()*, requires parameters identical to those of *xdr_bytes()* plus two more: the size of array elements, and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```
bool_t xdr_array(xdrs, ap, lp, maxlength, elementsize,
   xdr_element)
        XDR *xdrs;
        char **ap;
        u_int *lp;
        u_int maxlength;
        u_int elementsize;
        bool_t (*xdr_element)();
```

The parameter *ap* is the address of the pointer to the array. If **ap* is NULL when the array is being deserialized, XDR allocates an array of the appropriate size and sets **ap* to that array. The element count of the array is obtained from **lp* when the array is serialized; **lp* is set to the array length when the array is deserialized. The parameter *maxlength* is the maximum number of elements that the array is allowed to have; *elementsize* is the byte size of each element of the array (the C function *sizeof()* can be used to obtain this value). The *xdr_element()* routine is called to serialize, deserialize, or free each element of the array.

Examples

Before defining more constructed data types, consider the following examples.

Example A

A user on a networked machine can be identified by (a) the machine name, such as *krypton*: see *gethostname*(2); (b) the user's UID: see *geteuid*(2); and (c) the group numbers to which the user belongs: see *getgroups*(2). A structure with this information and its associated XDR routine could be coded like this:

```
struct netuser {
        char    *nu_machinename;
        int     nu_uid;
        u_int   nu_glen;
        int     *nu_gids;
};
#define NLEN 255 /* machine names must be shorter than 256 chars */
#define NGRPS 20 /* user can't be a member of more than 20 groups */

bool_t
xdr_netuser(xdrs, nup)
    XDR *xdrs;
    struct netuser *nup;
```

```
{
    return (xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
       xdr_int(xdrs, &nup->nu_uid) &&
       xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen, NGRPS,
          sizeof (int), xdr_int));
}
```

Example B

A party of network users could be implemented as an array of *netuser* structure. The declaration and its associated XDR routines are as follows:

```
struct party {
   u_int p_len;
   struct netuser *p_nusers;
};
#define PLEN 500 /* max number of users in a party */
bool_t
xdr_party(xdrs, pp)
   XDR *xdrs;
   struct party *pp;
{
   return (xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
            sizeof (struct netuser), xdr_netuser));
}
```

Example C

The well-known parameters to *main*, *argc* and *argv* can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like the following program.

```
struct cmd {
    u_int c_argc;
    char **c_argv;
};
#define ALEN 1000    /* args cannot be > 1000 chars */
#define NARGC 100    /* command cannot have > 100 args */

struct history {
    u_int h_len;
    struct cmd *h_cmds;
};
#define NCM  75      /* history is no more than 75 commands */

bool_t
xdr_wrap_string(xdrs, sp)
     XDR *xdrs;
     char **sp;
{
     return (xdr_string(xdrs, sp, ALEN));
}

bool_t
xdr_cmd(xdrs, cp)
     XDR *xdrs;
     struct cmd *cp;
{
     return (xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
        sizeof (char *), xdr_wrap_string));
}

bool_t
xdr_history(xdrs, hp)
    XDR *xdrs;
        struct history *hp;
{
    return (xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
        sizeof (struct cmd), xdr_cmd));
}
```

The most confusing part of this example is that the routine *xdr_wrap_string()* is needed to package the *xdr_string()* routine, because the implementation of *xdr_array()* only passes two parameters to the array element description routine; *xdr_wrap_string()* supplies the third parameter to *xdr_string*().

By now the recursive nature of the XDR library should be obvious. Continue with more constructed data types.

Opaque Data

In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The primitive *xdr_opaque()* is used for describing fixed sized, opaque bytes.

```
bool_t xdr_opaque(xdrs, p, len)
        XDR *xdrs;
        char *p;
        u_int len;
```

The parameter *p* is the location of the bytes; *len* is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object are not machine portable.

Fixed Size Arrays

The XDR library does not provide a primitive for fixed-length arrays (the primitive *xdr_array()* is for varying-length arrays). Example A could be rewritten to use fixed-sized arrays in the following way.

```
#define NLEN 255   /* machine names must be shorter than 256 chars */
#define NGRPS 20   /* user cannot be a member of more than 20 groups */

struct netuser {
    char *nu_machinename;
    int nu_uid;
    int nu_gids[NGRPS];
};


bool_t
xdr_netuser(xdrs, nup)
XDR *xdrs;
struct netuser *nup;
{
    int i;

    if (! xdr_string(xdrs, &nup->nu_machinename, NLEN))
            return (FALSE);
    if (! xdr_int(xdrs, &nup->nu_uid))
            return (FALSE);
    if (!xdr_vector(xdrs, nup->nu_gi , NGRPS, sizeof(int),
        xdr_int)) {
            return (FALSE);
    }
    return (TRUE);
}
```

6.3.6 Discriminated Unions

The XDR library supports discriminated unions. A discriminated union is a C union and an *enum_t* value that selects an "arm" of the union.

```
struct xdr_discrim {
        enum_t value;
        bool_t (*proc)();
};


bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
        XDR *xdrs;
        enum_t *dscmp;
        char *unp;
        struct xdr_discrim *arms;
        bool_t (*defaultarm)();   /* may equal NULL */
```

First the routine translates the discriminant of the union located at **dscmp*. The discriminant is always an *enum_t*. Next the union located at **unp* is

translated. The parameter *arms* is a pointer to an array of *xdr_discrim* structures. Each structure contains an ordered pair of *[value,proc]*. If the union's discriminant is equal to the associated *value*, then the *proc* is called to translate the union. The end of the *xdr_discrim* structure array is denoted by a routine of value NULL (0). If the discriminant is not found in the *arms* array, then the *defaultarm* procedure is called if it is non-NULL; otherwise the routine returns FALSE.

Example D

Suppose the type of a union may be integer, character pointer (a string), or a *gnumbers* structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };

struct u_tag {
    enum utype utype;    /* the union's discriminant */
        union {
                int ival;
                char *pval;
                struct gnumbers gn;
        } uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
    { INTEGER, xdr_int },
    { GNUMBERS, xdr_gnumbers }
    { STRING, xdr_wrap_string },
    { __dontcare__, NULL }
    /* always terminate arms with a NULL xdr_proc */
}


bool_t
xdr_u_tag(xdrs, utp)
    XDR *xdrs;
    struct u_tag *utp;
{
    return (xdr_union(xdrs, &utp->utype, &utp->uval, u_tag_arms,
            NULL));
}
```

The routine *xdr_gnumbers()* was presented above in "The XDR Library" section. *xdr_wrap_string()* was presented in Example C. The default *arm*

parameter to *xdr_union()* (the last parameter) is *NULL* in this example. Therefore the value of the union's discriminant may legally take on only values listed in the *u_tag_arms* array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

It is worth pointing out that the values of the discriminant may be sparse, though in this example they are not. It is always good practice to assign explicitly integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

Exercise: Implement *xdr_union()* using the other primitives in this section.

Pointers

In C it is often convenient to put pointers to another structure within a structure. The *xdr_reference()* primitive makes it easy to serialize, deserialize, and free these referenced structures.

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

Parameter *pp* is the address of the pointer to the structure; parameter *ssize* is the size in bytes of the structure (use the C function *sizeof()* to obtain this value); and *proc* is the XDR routine that describes the structure. When decoding data, storage is allocated if **pp* is NULL.

There is no need for a primitive *xdr_struct()* to describe structures within structures, because pointers are always sufficient.

Exercise: Implement *xdr_reference()* using *xdr_array*().

Caution: *xdr_reference()* and *xdr_array()* are **not** interchangeable external representations of data.

Example E

Suppose there is a structure containing a person's name and a pointer to a *gnumbers* structure containing the person's gross assets and liabilities. The next example shows this construct.

```
struct pgn {
    char *name;
    struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
        XDR *xdrs;
        struct pgn *pp;
{
    if (xdr_string(xdrs, &pp->name, NLEN) &&
        xdr_reference(xdrs, &pp->gnp,
                sizeof(struct gnumbers), xdr_gnumbers))
        return(TRUE);
    return(FALSE);
}
```

Pointer Semantics and XDR

In many applications, C programmers attach double meaning to the values of a pointer. Typically the value NULL (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For instance, in example E, a NULL pointer value for *gnp* could indicate that the person's assets and liabilities are unknown. That is, the pointer value encodes two things: whether or not the data is known; and if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive *xdr_reference()* cannot and does not attach any special meaning to a NULL-value pointer during serialization. That is, passing an address of a pointer whose value is NULL to *xdr_reference()* when serialing data will most likely cause a memory fault and, on the UNIX system, a core dump.

xdr_pointer() correctly handles NULL pointers. For more information about its use, see the *Linked Lists* topics below.

Exercise: After reading the section on *Linked Lists*, return here and extend example E so that it can correctly deal with NULL pointer values.

Exercise: Using the *xdr_union()*, *xdr_reference()* and *xdr_void()* primitives, implement a generic pointer handling primitive that implicitly deals with NULL pointers. That is, implement *xdr_pointer*().

6.3.7 Non-filter Primitives

XDR streams can be manipulated with the primitives discussed in this section.

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


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


xdr_destroy(xdrs)
        XDR *xdrs;
```

The routine *xdr_getpos()* returns an unsigned integer that describes the current position in the data stream.

Caution: In some XDR streams, the returned value of *xdr_getpos()* is meaningless; the routine returns a –1 in this case (though –1 should be a legitimate value).

The routine *xdr_setpos()* sets a stream position to *pos*. Warning: In some XDR streams, setting a position is impossible; in such cases, *xdr_setpos()* will return FALSE. This routine will also fail if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The *xdr_destroy()* primitive destroys the XDR stream. Usage of the stream after calling this routine is undefined.

6.4 XDR Operation Directions

At times you may wish to optimize XDR routines by taking advantage of the direction of the operation (XDR_ENCODE, XDR_DECODE, or XDR_FREE). The value *xdrs->x_op* always contains the direction of the XDR operation. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in the "Linked Lists" section later in this chapter, demonstrates the usefulness of the *xdrs->x_op* field.

6.5 XDR Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream.

Streams currently exist for (de)serialization of data to or from standard I/O FILE streams, TCP/IP connections and files, and memory. Section 6.6 documents the XDR object and how to make new XDR streams when they are required.

6.5.1 Standard I/O Streams

XDR streams can be interfaced to standard I/O using the *xdrstdio_create()* routine as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>      /* xdr streams part of rpc */

void
xdrstdio_create(xdrs, fp, x_op)
        XDR *xdrs;
        FILE *fp;
        enum xdr_op x_op;
```

The routine *xdrstdio_create()* initializes an XDR stream pointed to by *xdrs*. The XDR stream interfaces to the standard I/O library. Parameter *fp* is an open file, and *x_op* is an XDR direction.

6.5.2 Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory.

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

void
xdrmem_create(xdrs, addr, len, x_op)
        XDR *xdrs;
        char *addr;
        u_int len;
        enum xdr_op x_op;
```

The routine *xdrmem_create()* initializes an XDR stream in local memory. The memory is pointed to by parameter *addr*; parameter *len* is the length in bytes of the memory. The parameters *xdrs* and *x_op* are identical to the corresponding parameters of *xdrstdio_create*(). Currently, the UDP/IP implementation of RPC uses *xdrmem_create*(). Complete call or result messages are built in memory before calling the *sendto()* system routine.

6.5.3 Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the UNIX file or 4.2 BSD connection interface.

```
#include <rpc/rpc.h>   /* xdr streams are a part of the rpc library */

xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc,
   writeproc)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *iohandle;
        int (*readproc)(), (*writeproc)();
```

The routine *xdrrec_create()* provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX files.

The parameter *xdrs* is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters *sendsize* and *recvsize* determine the size in bytes of the

output and input buffers, respectively; if their values are zero (0), then predetermined defaults are used. When a buffer nee to be filled or flushed, the routine *readproc()* or *writeproc()* is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls *read()* and *write*(). However, the first parameter to each of these routines is the opaque parameter *iohandle*. The other two parameters *buf* and *nbytes*) and the results (byte count) are identical to the system routines. If *xxx* is *readproc()* or *writeproc*(), then it has the following form:

```
/*
/* Returns the actual number of bytes transferred.
 * -1 is an error
 */
int
xxx(iohandle, buf, len)
        char *iohandle;
        char *buf;
        int nbytes;
```

The XDR stream provides means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in the *Advanced Topics* topic below. The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
        XDR *xdrs;
        bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
        XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
        XDR *xdrs;
```

The routine *xdrrec_endofrecord()* causes the current outgoing data to be marked as a record. If the parameter *flushnow* is TRUE, then the stream's *writeproc()* will be called; otherwise, *writeproc* will be called when the output buffer has been filled.

The routine *xdrrec_skiprecord()* causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, the routine *xdrrec_eof()* returns TRUE, i.e., no more data in the underlying file descriptor.

6.6 XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

6.6.1 The XDR Object

The following structure defines the interface to an XDR stream:

```
enum xdr_op { XDR_ENCODE = 0, XDR_DECODE = 1, XDR_FREE = 2 };

typedef struct {
    enum xdr_op x_op;              /* operation; fast added param */
        struct xdr_ops {
        bool_t   (*x_getlong)();   /* get long from stream */
        bool_t   (*x_putlong)();   /* put long to stream */
        bool_t   (*x_getbytes)();  /* get bytes from stream */
        bool_t   (*x_putbytes)();  /* put bytes to stream */
        u_int    (*x_getpostn)();  /* return stream offset */
        bool_t   (*x_setpostn)();  /* reposition offset */
        caddr_t  (*x_inline)();    /* ptr to buffered data */
        VOID     (*x_destroy)();   /* free private area */
        } *x_ops;
    caddr_t      x_public;         /* users' data */
    caddr_t      x_private;        /* pointer to private data */
    caddr_t      x_base;           /* private for position info */
    int          x_handy;          /* extra private word */
} XDR;
```

The *x_op* field is the current operation being performed on the stream. This field is important to the XDR primitives, but should not affect a stream's implementation. That is, a stream's implementation should not depend on this value. The fields *x_private*, *x_base*, and *x_handy* are private to the particular stream's implementation. The field *x_public* is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives. *x_getpostn()*, *x_setpostn()* and *x_destroy()* are macros for accessing operations. The operation *x_inline()* takes two parameters: an XDR *, and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use

the buffer segment for any purpose. From the stream's point of view, the bytes in the buffer segment have been consumed or put. The routine may return NULL if it cannot return a buffer segment of the requested size. (The *x_inline()* routine is for cycle squeezers. Use of the resulting buffer is not data-portable. Users are encouraged not to use this feature.)

The operations *x_getbytes()* and *x_putbytes()* blindly get and put sequences of bytes from or to the underlying stream; they return TRUE if they are successful, and FALSE otherwise. The routines have identical parameters (replace *xxx*):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
        XDR *xdrs;
        char *buf;
        u_int bytecount;
```

The operations *x_getlong()* and *x_putlong()* receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The IRIX primitives *htonl()* and *ntohl()* can be helpful in accomplishing this. Section 6.8 defines the standard representation of numbers. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that nonnegative integers have the same bit representations as unsigned integers. The routines return TRUE if they succeed, and FALSE otherwise. They have identical parameters:

```
bool_t
xxxlong(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

Implementors of new XDR streams must make an XDR structure (with new operation routines) available to clients, using some kind of create routine.

6.7 Advanced Topics

This section describes additional techniques for passing data structures, e.g., linked lists (of arbitrary lengths). Unlike the simpler examples covered earlier, the following examples are written using both the XDR C library routines and the XDR data description language.

6.7.1 Linked Lists

The last example in the *Pointers* topic earlier in this chapter presented a C data structure and its associated XDR routines for a individual's gross assets and liabilities. The example is duplicated below:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};

bool_t
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long(xdrs, &(gp->g_assets)))
            return (xdr_long(xdrs, &(gp->g_liabilities)));
        return (FALSE);
}
```

Now assume that you wish to implement a linked list of such information. A data structure could be constructed as follows:

```
struct gnumbers_node {
        struct gnumbers gn_numbers;
        struct gnumbers_node *gn_next;
};

typedef struct gnumbers_node *gnumbers_list;
```

The head of the linked list can be thought of as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly the *gn_next* field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the *gn_next* field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive type declaration of *gnumbers_list*:

```
typedef union switch (boolean) {
        case TRUE: struct {
                struct gnumbers current_element;
                gnumbers_list rest_of_list;
        };
        case FALSE: struct {};
} gnumbers_list;
```

In this description, the boolean indicates whether there is more data following it. If the boolean is FALSE, then it is the last data field of the structure. If it is TRUE, then it is followed by a *gnumbers* structure and (recursively) by a *gnumbers_list* (the rest of the object). Note that the C declaration has no boolean explicitly declared in it (though the *gn_next* field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing the XDR routines for a *gnumbers_list* follow easily from the XDR description above. Note how the primitive *xdr_pointer()* is used to implement the XDR union above.

```
bool_t
xdr_gnumbers_node(xdrs, gn)
        XDR *xdrs;
        gnumbers_node *gn;
{
        return(xdr_gnumbers(xdrs, &gn->gn_numbers) &&
                xdr_gnumbers_list(xdrs, &gp->gn_next));
}

bool_t
xdr_gnumbers_list(xdrs, gnp)
        XDR *xdrs;
        gnumbers_list *gnp;
{
        return(xdr_pointer(xdrs, gnp,
                sizeof(struct gnumbers_node),
                xdr_gnumbers_node));
}
```

The unfortunate side effect of XDRing a list with these routines is that the C stack grows linearly with respect to the number of node in the list. This is due to the recursion. The following routine collapses the above two mutually recursive into a single, non-recursive one.

```
bool_t
xdr_gnumbers_list(xdrs, gnp)
        XDR *xdrs;
        gnumbers_list *gnp;
{
        bool_t more_data;
        gnumbers_list *nextp;

        for (;;) {
                more_data = (*gnp != NULL);
                if (!xdr_bool(xdrs, &more_data)) {
                        return(FALSE);
                }
                if (! more_data) {
                        break;
                }
                if (xdrs->x_op == XDR_FREE) {
                        nextp = &(*gnp)->gn_next;
                }
                if (!xdr_reference(xdrs, gnp,
                    sizeof(struct gnumbers_node), xdr_gnumbers)) {
                        return(FALSE);
                }
                gnp = (xdrs->x_op == XDR_FREE) ?
                        nextp : &(*gnp)->gn_next;
        }
        *gnp = NULL;
        return(TRUE);
}
```

The first task is to find out whether there is more data or not, so that this boolean information can be serialized. Notice that this statement is unnecessary in the XDR_DECODE case, since the value of more_data is not known until you deserialize it in the next statement.

The next statement XDR's the more_data field of the XDR union. If there is no more data, set this last pointer to NULL to indicate the end of the list, and return TRUE because you are done. Note that setting the pointer to NULL is only important in the XDR_DECODE case, since it is already NULL in the XDR_ENCODE and XDR_FREE cases.

Next, if the direction is XDR_FREE , the value of *nextp* is set to indicate the location of the next pointer in the list. You do this now because you need to dereference *gnp* to find the location of the next item in the list, and after the next statement the storage pointed to by *gnp* will be freed up and no be longer valid. You can't do this for all directions though, because in the XDR_DECODE direction the value of *gnp* won't be set until the next statement.

Next, XDR the data in the node using the primitive *xdr_reference*(). *xdr_reference()* is like *xdr_pointer()* which you used before, but it does not send over the boolean indicating whether there is more data. Use it instead of *xdr_pointer()* because you have already XDR'd this information ourselves. Notice that the XDR routine passed is not the same type as an element in the list. The routine passed is *xdr_gnumbers*(), for XDR'ing gnumbers, but each element in the list is actually of type *gnumbers_node*. You don't pass *xdr_gnumbers_node()* because it is recursive, and instead use *xdr_gnumbers()* which XDR's all of the non-recursive part. Note that this trick will work only if the *gn_numbers* field is the first item in each element, so that their addresses are identical when passed to *xdr_reference*().

Finally, update *gnp* to point to the next item in the list. If the direction is XDR_FREE , set it to the previously saved value, otherwise you can dereference *gnp* to get the proper value. Though harder to understand than the recursive version, this non-recursive routine is far less likely to blow the C stack. It will also run more efficiently since a lot of procedure call overhead has been removed. Most lists are small though (in the hundreds of items or less) and the recursive version should be sufficient for them.

6.8 XDR Specification

This section defines the External Data Representation (XDR) protocol specification. XDR is a standard for the description and encoding of data. It is useful for transferring data between different computer architectures, and has been used to communicate data between such diverse machines as the IRIS, Sun, VAX, IBM-PC, and Cray computers. XDR fits into the ISO presentation layer, and is roughly analogous in purpose to X.409, ISO Abstract Syntax Notation. The major difference between these two is that XDR uses implicit typing, while X.409 uses explicit typing.

XDR uses a language to describe data formats. The language can only be used only to describe data; it is not a programming language. This language allows one to describe intricate data formats in a concise manner. The alternative of using graphical representations (itself an informal language) quickly becomes incomprehensible when faced with complexity. The XDR language itself is similar to the C language, just as the Xerox Courier standard is similar to Mesa. Protocols such as Sun RPC (Remote Procedure

Call) and the NFS (Network File System) use XDR to describe the format of their data.

The XDR standard makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. It is assumed that hardware that encodes bytes onto various media will preserve the bytes' meanings across hardware boundaries. For example, the Ethernet standard suggests that bytes be encoded in "little-endian" style, or least significant bit first.

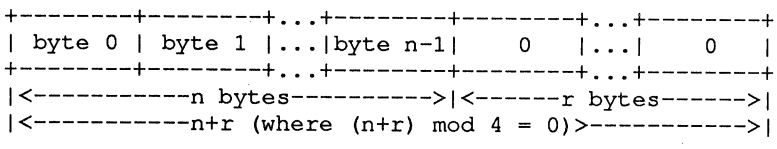
Once XDR data are shared among machines, it should not matter that the data was produced on an IRIS, but is consumed by a VAX (or vice versa). Similarly the choice of operating systems should have no influence on how the data is represented externally. For programming languages, data produced by a C program should be readable by a FORTRAN or Pascal program.

6.8.1 Basic Block Size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through n–1. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. If the n bytes needed to contain the data are not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of 4.

Include the familiar graphic box notation for illustration and comparison. In most illustrations, each box (delimited by a plus sign at the 4 corners and vertical bars and dashes) depicts a byte. Ellipses (...) between boxes show zero or more additional bytes where required.

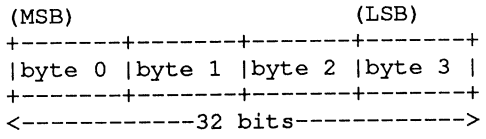
Block



6.8.2 Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range *[–2147483648, 2147483647]*. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. The data description of integers is *integer*.

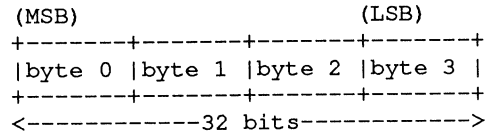
Integer



6.8.3 Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range *[0,4294967295]*. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. The data description of unsigned integers is *unsigned.*

Unsigned Integer



6.8.4 Enumerations

Enumerations have the same representation as integers. Enumerations are handy for describing subsets of the integers. The data description of enumerated data is as follows.

```
enum { name-identifier = constant, ... } identifier;
```

The three colors, e.g., red, yellow and blue could be described by an enumerated type.

```
enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

It is an error to encode as an enum any other integer than those that have been given assignments in the enum declaration.

6.8.5 Booleans

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Booleans are declared as follows:

```
bool identifier;
```

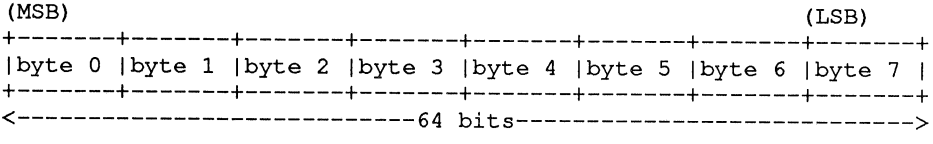
This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

6.8.6 Hyper Integer and Hyper Unsigned

The standard also defines 64-bit (8-byte) numbers called hyper integer and unsigned hyper integer. Their representations are the obvious extensions of integer and unsigned integer defined above. They are represented in two's complement notation. The most and least significant bytes are 0 and 7, respectively.

Hyper Integer
Unsigned Hyper Integer



6.8.7 Floating Point

The standard defines the floating-point data type "float" (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating point numbers. See the ANSI/IEEE 754-1985 floating point standard for more information.

The following three fields describe the single-precision floating-point number:

S The sign of the number. Values 0 and 1 represent positive and negative, respectively.

E The exponent of the number, base 2. 8 bits are devoted to this field. The exponent is biased by 127.

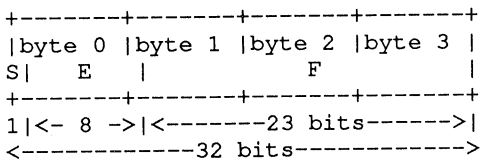
F The fractional part of the number's mantissa, base 2. 23 bits are devoted to this field.

Therefore, the floating point number is described by:

$$(-1)^{S} * 2^{(E - Bias)} * 1.F$$

It is declared as follows:

Single-Precision Floating-Point



Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating- point number are 0 and 31. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 9, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

6.8.8 Double-precision Floating-point

The standard defines the encoding for the double-precision floating- point data type "double" (64 bits or 8 bytes). The encoding used is the ANSI/IEEE 754/1985 standard for normalized double-precision floating-point numbers. The standard encodes the following three fields, which describe the double-precision floating-point number:

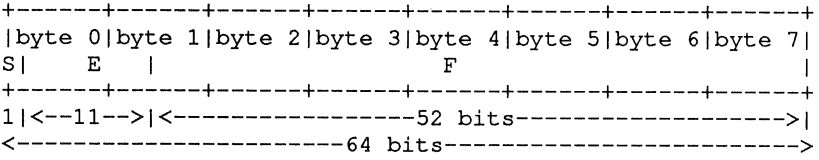
S The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E The exponent of the number, base 2. 11 bits are devoted to this field. The exponent is biased by 1023.

F The fractional part of the number's mantissa, base 2. 52 bits are devoted to this field.

It is declared as follows:

Double-Precision Floating-Point



Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a double-precision floating- point number are 0 and 63. The beginning bit (and most significant bit) offsets of S, E , and F are 0, 1, and 12, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the "NaN" (not a number) is system dependent and should not be used externally.

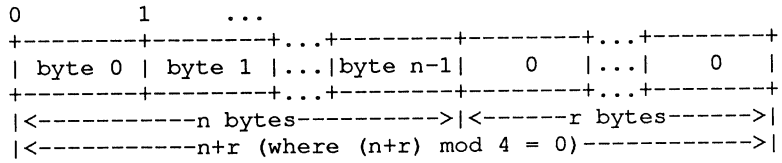
6.8.9 Fixed-Length Opaque Data

At times, fixed-sized uninterpreted data needs to be passed among machines. This data is called "opaque" and is declared as follows:

```
opaque identifier[n];
```

where the constant n is the (static) number of bytes necessary to contain the opaque data. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count of the opaque object a multiple of four.

Fixed-Length Opaque



6.8.10 Variable-length Opaque Data

The standard also provides for variable-length (counted) opaque data, defined as a sequence of n (numbered 0 through n–1) arbitrary bytes to be the number n encoded as an unsigned integer (as described below), and followed by the n bytes of the sequence.

Byte m of the sequence always precedes byte m+1 of the sequence, and byte 0 of the sequence always follows the sequence's length (count). enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Variable-length opaque data is declared in the following way:

```
opaque identifier<m>;
```

or

```
opaque identifier<>;
```

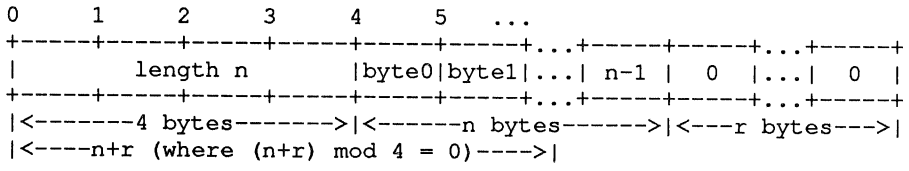
The constant m denotes an upper bound of the number of bytes that the sequence may contain. If m is not specified, as in the second declaration, it is assumed to be $(2^{32}) - 1$, the maximum length. The constant m would

normally be found in a protocol specification. For example, a filing protocol may state that the maximum data transfer size is 8192 bytes, as follows:

```
opaque filedata<8192>;
```

It is an error to encode a length greater than the maximum described in the specification.

Variable-Length Opaque



6.8.11 String

The standard defines a string of n (numbered 0 through n–1) ASCII bytes to be the number n encoded as an unsigned integer (as described above), and followed by the n bytes of the string. Byte m of the string always precedes byte m+1 of the string, and byte 0 of the string always follows the string's length. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Counted byte strings are declared as follows:

```
string object<m>;
```

or

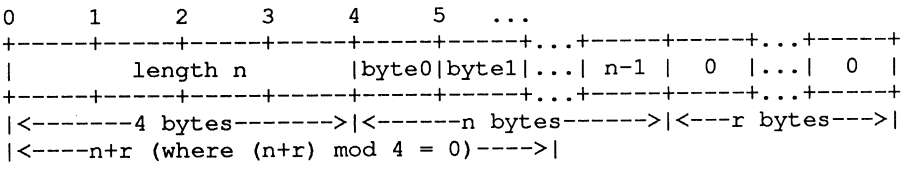
```
string object<>;
```

The constant m denotes an upper bound of the number of bytes that a string may contain. If m is not specified, as in the second declaration, it is assumed to be $(2^{32}) - 1$, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, as follows:

```
string filename<255>;
```

It is an error to encode a length greater than the maximum described in the specification.

A String



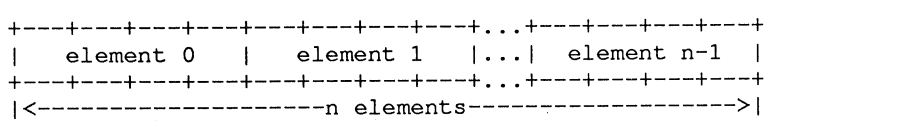
6.8.12 Fixed-length Array

Declarations for fixed-length arrays of homogeneous elements are in the following form:

```
type-name identifier[n];
```

Fixed-length arrays of elements numbered 0 through n–1 are encoded by individually encoding the elements of the array in their natural order, 0 through n–1. Each element's size is a multiple of four bytes. Though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of type "string", yet each element will vary in its length.

Fixed-Length Array



6.8.13 Variable-length Array

Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count n (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element n- 1.

The declaration for variable-length arrays follows this form:

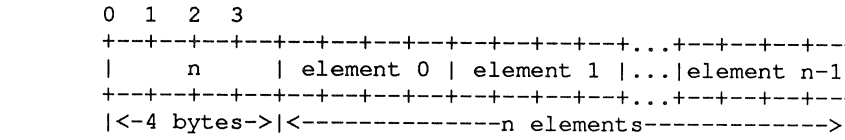
```
type-name identifier<m>;
```

or

```
type-name identifier<>;
```

The constant m specifies the maximum acceptable element count of an array; if m is not specified, as in the second declaration, it is assumed to be $(2^{32}) - 1$. It is an error to encode a value of n that is greater than the maximum described in the specification.

Counted Array



6.8.14 Structures

The data description for structures is very similar to that of standard C:

```
struct {
        component-declaration-A;
        component-declaration-B;
        ...
} identifier;
```

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, though the components may be different sizes.

6.8.15 Discriminated Unions

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either *int*, *unsigned int*, or an

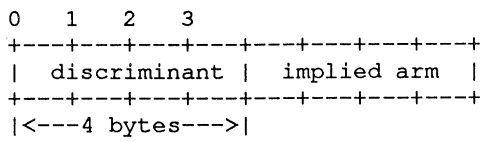
enumerated type, such as *bool*. The component types are called "arms" of the union, and are preceded by the value of the discriminant which implies their encoding.

Discriminated unions are declared as shown in the next example.

```
union switch (discriminant-declaration) {
        case discriminant-value-A:
        arm-declaration-A;
        case discriminant-value-B:
        arm-declaration-B;
        ...
        default: default-declaration;
} identifier;
```

Each "case" keyword is followed by a legal value of the discriminant. The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

Discriminated Union

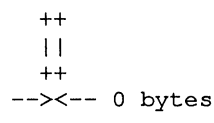


6.8.16 Void

An XDR void is a 0-byte quantity. Voids are useful for describing operations that take no data as input or no data as output. They are also useful in unions, where some arms may contain data and others do not. The declaration is simply as follows:

```
void;
```

Void



6.8.17 Constant

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

"const" is used to define a symbolic name for a constant; it does not declare any data. The symbolic constant may be used anywhere a regular constant may be used. For example, the following defines a symbolic constant DOZEN, equal to 12.

```
const DOZEN = 12;
```

6.8.18 Typedef

typedef does not declare any data either, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the *typedef*. For example, the following defines a new type called "eggbox" using an existing type called "egg":

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the *typedef*, if it was considered a variable. For example, the following two declarations are equivalent in declaring the variable "fresheggs":

```
eggbox  fresheggs;
egg     fresheggs[DOZEN];
```

When a *typedef* involves a struct, enum, or union definition, there is another (preferred) syntax that may be used to define the same type.

In general, a *typedef* of the following form:

```
typedef <<struct, union, or enum definition>> identifier;
```

may be converted to the alternative form by removing the *typedef* part and placing the identifier after the *struct*, *union*, or *enum* keyword, instead of at the end. For example, there are two ways to define the type *bool*.

```
typedef enum {      /* using typedef */
        FALSE = 0,
        TRUE = 1
} bool;

enum bool {         /* preferred alternative */
        FALSE = 0,
        TRUE = 1
};
```

The reason this syntax is preferred is one does not have to wait until the end of a declaration to figure out the name of the new type.

6.8.19 Optional-data

Optional-data is one kind of union that occurs so frequently that we give it a special syntax of its own for declaring it. It is declared as follows:

```
type-name *identifier;
```

This is equivalent to the following union:

```
union switch (bool opted) {
        case TRUE:
            type-name element;
        case FALSE:
            void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, since the boolean "opted" can be interpreted as the length of the array:

```
type-name identifier<1>;
```

Optional-data is very useful for describing recursive data-structures such as linked-lists and trees. The following example defines a type "stringlist" that encodes lists of arbitrary length strings.

```
struct *stringlist {
        string item<>;
        stringlist next;
};
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
        case TRUE:
                struct {
                        string item<>;
                        stringlist next;
                } element;
        case FALSE:
                void;
};
```

or as a variable-length array:

```
struct stringlist<1> {
        string item<>;
        stringlist next;
};
```

Both of these declarations obscure the intention of the stringlist type, so the optional-data declaration is preferred over both of them. The optional-data type also has a close correlation to how recursive data structures are represented in high-level languages such as Pascal or C by use of pointers. In fact, the syntax is the same as that of the C language for pointers.

6.8.20 Areas for Future Enhancement

The XDR standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. Also missing are packed (or binary-coded) decimals.

The intent of the XDR standard was not to describe every kind of data that people have ever sent or will ever want to send from machine to machine. Rather, it only describes the most commonly used data-types of high-level languages such as Pascal or C so that applications written in these languages will be able to communicate easily over some medium.

One could imagine extensions to XDR that would let it describe almost any existing protocol, such as TCP. The minimum necessary for this are support

for different block sizes and byte-orders. The XDR discussed here could then be considered the 4-byte big-endian member of a larger XDR family.

6.8.21 Discussion

The following subsections may answer some of your XDR questions.

Why Have a Language for Describing Data?

There are many advantages in using a data-description language such as XDR versus using diagrams. Languages are more formal than diagrams and lead to less ambiguous descriptions of data. Languages are also easier to understand and allow one to think of other issues instead of the low-level details of bit-encoding. Also, there is a close analogy between the types of XDR and a high-level language such as C or Pascal. This makes the implementation of XDR encoding and decoding modules an easier task. Finally, the language specification itself is an ASCII string that can be passed from machine to machine to perform on-the-fly data interpretation.

Why Is There Only One Byte-Order for an XDR Unit?

Supporting two byte-orderings requires a higher level protocol for determining in which byte-order the data is encoded. Since XDR is not a protocol, this can't be done. The advantage of this, though, is that data in XDR format can be written to a magnetic tape, for example, and any machine will be able to interpret it, since no higher level protocol is necessary for determining the byte-order.

Why Does XDR Use Big-Endian Byte-Order?

Yes, it is unfair, but having only one byte-order means you have to be unfair to somebody. Many architectures, such as the MIPS R2000/3000, Motorola 68000 and IBM 370, support the big-endian byte-order.

Why Is the XDR Unit Four Bytes Wide?

There is a tradeoff in choosing the XDR unit size. Choosing a small size such as two makes the encoded data small, but causes alignment problems for machines that aren't aligned on these boundaries. A large size such as eight means the data will be aligned on virtually every machine, but causes the encoded data to grow too big. Select four as a compromise. Four is big enough to support most architectures efficiently, except for rare machines such as the eight-byte aligned Cray. Four is also small enough to keep the encoded data restricted to a reasonable size.

Why Must Variable-Length Data Be Padded with Zeros?

It is desirable that the same data encode into the same thing on all machines, so that encoded data can be meaningfully compared or checksummed. Forcing the padded bytes to be zero ensures this.

Why Is There No Explicit Data-Typing?

Data-typing has a relatively high cost for what small advantages it may have. One cost is the expansion of data due to the inserted type fields. Another is the added cost of interpreting these type fields and acting accordingly. And most protocols already know what type they expect, so data-typing supplies only redundant information. However, one can still get the benefits of data-typing using XDR. One way is to encode two things: first a string which is the XDR data description of the encoded data, and then the encoded data itself. Another way is to assign a value to all the types in XDR, and then define a universal type which takes this value as its discriminant and for each value, describes the corresponding data type.

6.9 The XDR Language Specification

This section describes the XDR language specification including:

- notational conventions
- lexical notes
- syntax information and notes

6.9.1 Notational Conventions

This specification uses an extended Backus-Naur Form notation for describing the XDR language. This notation includes:

1. Special characters:

   | ( ) [ ] " *

2. Terminal symbols are strings of any characters surrounded by double quotes.
3. Non-terminal symbols are strings of non-special characters.
4. Alternative items are separated by a vertical bar (|).
5. Optional items are enclosed in brackets.
6. Items are grouped together by enclosing them in parentheses.
7. A * following an item means 0 or more occurrences of that item.

For example, consider the following pattern:

"a " "very" (", " " very")* [" cold " "and"] " rainy " ("day" | "night")

An infinite number of strings match this pattern; some are:

"a very rainy day"
"a very, very rainy day"
"a very cold and rainy day"
"a very, very, very cold and rainy night"

6.9.2 Lexical Notes

1. Comments begin and end with /* *comment* */, respectively.
2. White space serves to separate items and is otherwise ignored.
3. An identifier is a letter followed by an optional sequence of letters, digits or underbar ('_'). The case of identifiers is not ignored.
4. A constant is a sequence of one or more decimal digits, optionally preceded by a minus-sign ('–').

6.9.3 Syntax Information

```
declaration:
        type-specifier identifier
        | type-specifier identifier "[" value "]"
        | type-specifier identifier "<" [ value ] ">"
        | "opaque" identifier "[" value "]"
        | "opaque" identifier "<" [ value ] ">"
        | "string" identifier "<" [ value ] ">"
        | type-specifier "*" identifier
        | "void"
value:
        constant
        | identifier

type-specifier:
          [ "unsigned" ] "int"
        | [ "unsigned" ] "hyper"
        | "float"
        | "double"
        | "bool"
        | enum-type-spec
        | struct-type-spec
        | union-type-spec
        | identifier

enum-type-spec:
        "enum" enum-body

enum-body:
        "{"
        ( identifier "=" value )
        ( "," identifier "=" value )*
        "}"
```

```
struct-type-spec:
        "struct" struct-body

struct-body:
        "{"
        ( declaration ";" )
        ( declaration ";" )*
        "}"

union-type-spec:
        "union" union-body

union-body:
        "switch" "(" declaration ")" "{"
        ( "case" value ":" declaration ";" )
        ( "case" value ":" declaration ";" )*
        [ "default" ":" declaration ";" ]
        "}"

constant-def:
        "const" identifier "=" constant ";"

type-def:
        "typedef" declaration ";"
        | "enum" identifier enum-body ";"
        | "struct" identifier struct-body ";"
        | "union" identifier union-body ";"

definition:
        type-def
        | constant-def

specification:
        definition *
```

6.9.4 Syntax Notes

1. The following are keywords and cannot be used as identifiers: "bool", "case", "const", "default", "double", "enum", "float", "hyper", "opaque", "string", "struct", "switch", "typedef", "union", "unsigned" and "void".
2. Only unsigned constants may be used as size specifications for arrays. If an identifier is used, it must have been declared previously as an unsigned constant in a "const" definition.
3. Constant and type identifiers within the scope of a specification are in the same name space and must be declared uniquely within this scope.

4. Similarly, variable names must be unique within the scope of struct and union declarations. Nested struct and union declarations create new scopes.

5. The discriminant of a union must be of a type that evaluates to an integer. That is, "int", "unsigned int", "bool", an enumerated type or any typedefed type that evaluates to one of these is legal. Also, the case values must be one of the legal values of the discriminant. Finally, a case value may not be specified more than once within the scope of a union declaration.

6.9.5 XDR Data Description Example

Following is a short XDR data description of "file" that you might use to transfer files from one machine to another.

```
const MAXUSERNAME = 32;     /* max length of a user name */
const MAXFILELEN = 65535;   /* max length of a file     */
const MAXNAMELEN = 255;     /* max length of a file name */

/*
 * Types of files:
 */

enum filekind {
        TEXT = 0,       /* ascii data */
        DATA = 1,       /* raw data  */
        EXEC = 2        /* executable */
};
/*
 * File information, per kind of file:
 */

union filetype switch (filekind kind) {
    case TEXT:
        void;                                   /* no extra information */
    case DATA:
        string creator<MAXNAMELEN>;             /* data creator */
    case EXEC:
        string interpretor<MAXNAMELEN>; /* program interpretor */
};

/*
 * A complete file:
 */

struct file {
```

```
        string filename<MAXNAMELEN>;  /* name of file */
        filetype type;                /* info about file */
        string owner<MAXUSERNAME>;    /* owner of file  */
        opaque data<MAXFILELEN>;      /* file data      */
};
```

Suppose now that there is a user named "john" who wants to store his lisp program "sillyprog" that contains just the data "(quit)". His file would be encoded as follows:

Offset	Hex Bytes	ASCII	Description
0	00 00 00 09	....	Length of filename = 9
4	73 69 6c 6c	sill	Filename characters
8	79 70 72 6f	ypro	... and more characters ...
12	67 00 00 00	g...	... and 3 zero-bytes of fill
16	00 00 00 02	....	Filekind is EXEC = 2
20	00 00 00 04	....	Length of interpreter = 4
24	6c 69 73 70	lisp	Interpreter characters
28	00 00 00 04	....	Length of owner = 4
32	6a 6f 68 6e	john	Owner characters
36	00 00 00 06	....	Length of file data = 6
40	28 71 75 69	(qui	File data bytes ...
44	74 29 00 00	t)..	... and 2 zero-bytes of fill

7. RPC Specification

This chapter assumes that you are familiar with both RPC and XDR as described in Chapters 4, 5 and 6. It does not attempt to justify RPC or its uses. Also, the casual user of RPC does not need to be familiar with the information in this chapter.

7.1 Introduction

This chapter specifies a message protocol used in implementing Sun's Remote Procedure Call (RPC) message protocol package. The message protocol is specified with the External Data Representation (XDR) language.

Topics covered in this chapter include:

- RPC protocol requirements
- authentication
- RPC message protocol
- port mapper program protocol

7.1.1 Terminology

The document discusses servers, services, programs, procedures, clients and versions. A server is a machine where some number of network services are implemented. A service is a collection of one or more remote programs. A remote program implements one or more remote procedures; the procedures, their parameters and results are documented in the specific program's protocol specification. Network clients are pieces of software that initiate

remote procedure calls to services. A server may support more than one version of a remote program in order to be forward compatible with changing protocols.

For example, a network file service may be composed of two programs. One program may deal with high-level applications such as file system access control and locking. The other may deal with low-level file I/O, and have procedures like "read" and "write." A client machine of the network file service would call the procedures associated with the two programs of the service on behalf of some user on the client machine.

7.1.2 The RPC Model

The remote procedure call model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location (such as a result register). It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from the well-specified location, and the caller continues execution.

The remote procedure call is similar, except that one thread of control winds through two processes — one is the caller's process, the other is a server's process. That is, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message. Note that in this model, only one of the two processes is active at any given time. However, this model is only given as an example. The RPC protocol makes no restrictions on the concurrency model implemented, and others are possible. For example, an implementation may choose to have RPC calls be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a task to process an incoming request, so that the server can be free to receive other requests.

7.1.3 Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol only deals with the specification and interpretation of messages.

It is important to point out that RPC does not try to implement any kind of reliability and that the application must be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP/IP, then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP/IP, it must implement is own retransmission and time-out policy as the RPC layer does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport such as UDP/IP. If an application retransmits RPC messages after short time-outs, the only thing it can infer if it receives no reply is that the procedure was executed zero or more times. If it does receive a reply, then it can infer that the procedure was executed at least once.

A server may wish to remember previously granted requests from a client and not regrant them in order to ensure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request. The main use of this transaction is by the client RPC layer in matching replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, knowing this fact, may choose to remember this ID after granting a request and not regrant requests with the same ID in order to achieve some degree of execute-at-most-once semantics. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a reliable transport such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

There are other possibilities for transports besides datagram- or connection-oriented protocols. On IRIX, RPC is currently implemented on top of both TCP/IP and UDP/IP transports.

7.1.4 Binding and Rendezvous Independence

The act of binding a client to a service is NOT part of the remote procedure call specification. This important and necessary function is left up to some higher-level software. (The software may use RPC itself—see "Port Mapper Program Protocol," later in this chapter).

Implementors should think of the RPC protocol as the jump-subroutine instruction (JSR) of a network; the loader (binder) makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the network makes RPC useful, using RPC to accomplish this task.

7.1.5 Message Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice-versa. Security and access control mechanisms can be built on top of the message authentication. Several different authentication protocols can be supported. A field in the RPC header indicates which protocol is being used. More information on specific authentication protocols can be found in the "Authentication Protocols," later in this chapter.

7.2 RPC Protocol Requirements

The RPC protocol must provide for the following:

- unique specification of a procedure to be called
- provisions for matching response messages to request messages
- provisions for authenticating the caller to service and vice versa

Besides these requirements, features that detect the following are worth supporting because of protocol roll-over errors, implementation bugs, user error, and network administration:

- RPC protocol mismatches
- remote program protocol version mismatches
- protocol errors (like misspecification of a procedure's parameters)
- reasons why remote authentication failed
- any other reasons why the desired procedure was not called

7.2.1 Remote Programs and Procedures

The RPC call message has three unsigned fields: remote program number, remote program version number, and remote procedure number. The three fields uniquely identify the procedure to be called. Program numbers are administered by some central authority (like Sun Microsystems) Once an implementor has a program number, he can implement his remote program; the first implementation would most likely have the version number of 1. Because most new protocols evolve into better, stable and mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification may state that its procedure number 5 is `read` and procedure number 12 is `write`.

Just as remote program protocols 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; this field must be two (2) for the version of RPC described here.

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

- The remote implementation of RPC does speak protocol version 2. The lowest and highest supported RPC version numbers are returned.

- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist (this is usually a caller-side protocol or programming error).
- The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is caused by a disagreement about the protocol between client and service.)

7.3 Authentication

Provisions for authentication of caller to service and vice versa are provided as a wart on the side of the RPC protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

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

struct opaque_auth {
    auth_flavor flavor;
    opaque body<400>;
};
```

In simple English, any *opaque_auth* structure is an *auth_flavor* enumeration followed by bytes which are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. (See "Authentication Protocols," which follows, for definitions of the various authentication protocols.)

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

7.3.1 Program Number Assignment

See Chapter 4 for the RPC program number assignments.

7.4 Other Uses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message passing protocol with which other (non-RPC) protocols can be implemented. Sun currently uses (abuses) the RPC message protocol for the following two (non-RPC) protocols: batching (or pipelining) and broadcast RPC. These two protocols are discussed (but not defined) below.

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching uses reliable byte stream protocols (like TCP/IP) for its transport. In the case of batching, the client never waits for a reply from the server and the server does not send replies to batch requests. A sequence of batch calls is usually terminated by a legitimate RPC in order to flush the pipeline (with positive acknowledgement).

Broadcast RPC

In broadcast RPC based protocols, the client sends an a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet-based protocols (like UDP/IP) as its transport. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors. Broadcast RPC uses the Port Mapper RPC service to achieve its semantics. See "Port Mapper Program Protocol" later in this chapter for more information.

7.5 The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language specified in Chapter 6. The message is defined in a top-down style. Note that this is an XDR specification, not C code.

```
enum msg_type {
        CALL = 0,
        REPLY = 1
};


/*
 * A reply to a call message can take on two forms:
 * the message was either accepted or rejected.
 */
enum reply_stat {
        MSG_ACCEPTED = 0,
        MSG_DENIED = 1
};


/*
 * Given that a call message was accepted, the following is
 * the status of an attempt to call a remote procedure.
 */
enum accept_stat {
   /* remote procedure was successfully executed */
   SUCCESS=0,
   /* remote machine exports the program number */
   PROG_UNAVAIL=1,
   /* remote machine can't support version number */
   PROG_MISMATCH=2,
   /* remote program doesn't know about procedure */
   PROC_UNAVAIL=3,
   /* remote procedure can't figure out parameters */
   GARBAGE_ARGS=4
};
```

```
/*
 * Reasons why a call message was rejected:
 */
enum reject_stat {
   /* RPC version number was not two (2) */
   RPC_MISMATCH = 0,
   /* caller not authenticated on remote machine */
   AUTH_ERROR = 1
};


/*
 * Why authentication failed:
 */
enum auth_stat {
  AUTH_BADCRED = 1, /* bogus credentials (seal broken) */
  AUTH_REJECTEDCRED = 2, /* client should begin new session */
  AUTH_BADVERF = 3, /* bogus verifier (seal broken) */
  AUTH_REJECTEDVERF = 4, /* verifier expired or was replayed */
  AUTH_TOOWEAK = 5, /* rejected due to security reasons */
};


/*
 * The RPC message:
 * All messages start with a transaction identifier, xid,
 * followed by a two-armed discriminated union.  The
 * union's discriminant is a msg_type which switches to
 * one of the two types of the message.  The xid of a
 * REPLY message always matches that of the initiating
 * CALL message. NB: The xid field is only used for clients
 * matching reply messages with call messages or for servers
 * detecting  retransmissions; the service side cannot treat
 * this id as any type of sequence number.
 */
struct rpc_msg {
        unsigned int    xid;
        union switch (msg_type mtype) {
                case CALL:
                        call_body cbody;
                case REPLY:
                        reply_body rbody;
        } body;
};
```

```
/*
 * Body of an RPC request call:
 * In version 2 of the RPC protocol specification, rpcvers must
 * be equal to 2.  The fields prog, vers, and proc specify the
 * remote program, its version, and the procedure within the
 * remote program to be called.  These fields are followed by
 * two authentication parameters, cred (authentication
 * credentials) and verf (authentication verifier).  The two
 * authentication parameters are followed by the parameters to
 * the remote procedure, which are specified by the specific
 * program protocol.
 */
struct call_body {
        unsigned int rpcvers;   /* must be equal to two (2) */
        unsigned int prog;
        unsigned int vers;
        unsigned int proc;
        opaque_auth cred;
        opaque_auth verf;
        /* procedure specific parameters start here */
};


/*
 * Body of a reply to an RPC request.
 * The call message was either accepted or rejected.
 */
union reply_body switch (reply_stat stat) {
        case MSG_ACCEPTED:
                accepted_reply areply;
        case MSG_DENIED:
                rejected_reply rreply;
} reply;
```

```
/*
 * Reply to an RPC request that was accepted by the server.
 * Note: there could be an error even though the request
 * was accepted.  The first field is an authentication
 * verifier which the server generates in order to validate
 * itself to the caller.  It is followed by a union whose
 * discriminant is an enum accept_stat.  The SUCCESS arm of
 * the union is protocol specific.  The PROG_UNAVAIL,
 * PROC_UNAVAIL, and GARBAGE_ARGS arms of the union are
 * void.  The PROG_MISMATCH arm specifies the lowest and
 * highest version numbers of the remote program that are
 * supported by the server.
 */
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
            case SUCCESS:
                opaque results[0];
                /* procedure-specific results start here */
            case PROG_MISMATCH:
                struct {
                        unsigned int low;
                        unsigned int high;
                } mismatch_info;
            default:
                /*
                 * Void.  Cases include PROG_UNAVAIL,
                 * PROC_UNAVAIL, and GARBAGE_ARGS.
                 */
                void;
    } reply_data;
};
```

```
/*
 * Reply to an RPC request that was rejected by the server.
 * The request can be rejected because of two reasons: either
 * the server is not running a compatible version of the
 * RPC protocol (RPC_MISMATCH), or the server refused to
 * authenticate the caller (AUTH_ERROR). In the case of
 * an RPC version mismatch, the server returns the lowest and
 * highest supported RPC version numbers. In the case of
 * refused authentication, the failure status is returned.
 */
union rejected_reply switch (reject_stat stat) {
        case RPC_MISMATCH:
                struct {
                        unsigned int low;
                        unsigned int high;
                } mismatch_info;
        case AUTH_ERROR:
                auth_stat stat;
};
```

7.6 Authentication Protocols

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some "flavors" of authentication in this implementation. Other sites are free to invent new authentication types, with the same rules of flavor number assignment as there is for program number assignment.

7.6.1 Null Authentication

Often calls must be made where the caller does not know who he is and the server does not care who the caller is. In this case, the *auth_flavor* value (the discriminant of the *opaque_auth*'s union) of the RPC message's credentials, verifier, and response verifier is AUTH_NULL (0). The bytes of the opaque_auth's body are undefined. It is recommended that the opaque length be zero.

7.6.2 UNIX Authentication

The caller of a remote procedure may wish to identify himself as he is identified on a UNIX system. The value of the *credential's* discriminant of an RPC call message is AUTH_UNIX (1). The bytes of the *credential's* opaque body encode the the following structure:

```
struct auth_unix {
        unsigned int stamp;
        string machinename<255>;
        unsigned int uid;
        unsigned int gid;
        unsigned int gids<10>;
};
```

The *stamp* is an arbitrary id which the caller machine may generate. The *machinename* is the name of the caller's machine (like "krypton"). The *uid* is the caller's effective user id. The *gid* is the callers effective group id. The *gids* is a counted array of groups which contain the caller as a member. The *verifier* accompanying the credentials should be of AUTH_NULL (defined above).

The value of the discriminate of the "response verifier" received in the reply message from the server may be AUTH_NULL or AUTH_SHORT (2). In the case of AUTH_SHORT, the bytes of the *response verifier*'s string encode an opaque structure. This new opaque structure may now be passed to the server instead of the original AUTH_UNIX flavor credentials. The server keeps a cache which maps shorthand opaque structures (passed back via a AUTH_SHORT style "response verifier") to the original credentials of the caller. The caller can save network bandwidth and server CPU cycles by using the new credentials.

The server may flush the shorthand opaque structure at any time. If this happens, the remote procedure call message will be rejected due to an authentication error. The reason for the failure will be AUTH_REJECTEDCRED. At this point, the caller may wish to try the original AUTH_UNIX style of credentials.

7.7 Record Marking Standard

When RPC messages are passed on top of a byte stream protocol (like TCP/IP), it is necessary, or at least desirable, to delimit one message from another in order to detect and possibly recover from user protocol errors. This is called record marking (RM). This implementation of RPC uses this RM/TCP/IP transport for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by *0* to $2^{31}-1$ bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values — a boolean which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the highest-order bit of the header; the length is the 31 low-order bits. (Note that this record specification is *not* in XDR standard form!)

7.8 Port Mapper Program Protocol

The port mapper program maps RPC program and version numbers transport-specific port numbers. This program makes dynamic binding of remote programs possible.

This is desirable because the range of reserved port numbers is very small and the number of potential remote programs is very large. By running only the port mapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the port mapper.

The port mapper also aids in broadcast RPC. A given RPC program will usually have different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The port mapper, however, does have a fixed port number. So, to broadcast to a given program, the client actually sends its message to the port mapper located at the broadcast address. Each port mapper that picks up the broadcast then calls the local service specified by the client. When the port mapper gets the reply from the local service, it sends the reply on back to the client.

7.8.1 Port Mapper Protocol Specification (in RPC Language)

```
const PMAP_PORT = 111;          /* portmapper port number */


/*
 * A mapping of (program, version, protocol) to port number
 */
struct mapping {
        unsigned int prog;
        unsigned int vers;
        unsigned int prot;
        unsigned int port;
};

/*
 * Supported values for the "prot" field
 */
const IPPROTO_TCP = 6;          /* protocol number for TCP/IP */
const IPPROTO_UDP = 17;         /* protocol number for UDP/IP */

/*
 * A list of mappings
 */
struct *pmaplist {
        mapping map;
        pmaplist next;
};


/*
 * Arguments to callit
 */
struct call_args {
        unsigned int prog;
        unsigned int vers;
        unsigned int proc;
        opaque args<>;
};

/*
 * Results of callit
 */
struct call_result {
        unsigned int port;
        opaque res<>;
};
```

```
/*
 * Port mapper procedures
 */
program PMAP_PROG {
        version PMAP_VERS {
                void
                PMAPPROC_NULL(void)          = 0;

                bool
                PMAPPROC_SET(mapping)        = 1;

                bool
                PMAPPROC_UNSET(mapping)      = 2;

                unsigned int
                PMAPPROC_GETPORT(mapping)    = 3;

                pmaplist
                PMAPPROC_DUMP(void)          = 4;

                call_result
                PMAPPROC_CALLIT(call_args)   = 5;
        } = 2;
} = 100000;
```

7.8.2 Port Mapper Operation

The portmapper program currently supports two protocols (UDP/IP and TCP/IP). The portmapper is contacted by talking to it on assigned port number 111 ("sunrpc") on either of these protocols. The following is a description of each of the portmapper procedures:

PMAPPROC_NULL:

This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results. When a program first becomes available on a machine, it registers itself with the port mapper program on the same machine. The program passes its program number "prog", version number "vers", transport protocol number "prot", and the port "port" on which it awaits service request. The procedure returns a boolean response whose value is *TRUE* if the procedure successfully established the mapping and *FALSE* otherwise. The procedure refuses to establish a mapping if one already exists for the tuple "(prog, vers, prot)".

PMAPPROC_UNSET:

When a program becomes unavailable, it should unregister itself with the port mapper program on the same machine. The parameters and results have meanings identical to those of *PMAPPROC_SET*. The protocol and port number fields of the argument are ignored.

PMAPPROC_GETPORT:

Given a program number "prog", version number "vers", and transport protocol number "prot", this procedure returns the port number on which the program is awaiting call requests. A port value of zero means the program has not been registered. The "port" field of the argument is ignored.

PMAPPROC_DUMP:

This procedure enumerates all entries in the port mapper's database. The procedure takes no parameters and returns a list of program, version, protocol, and port values.

PMAPPROC_CALLIT:

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. It is intended for supporting broadcasts to arbitrary remote programs via the well-known port mapper's port. The parameters "prog", "vers", "proc", and the bytes of "args" are the program number, version number, procedure number, and parameters of the remote procedure.

Note: This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise.

The port mapper communicates with the remote program using UDP/IP only.

The procedure returns the remote program's port number, and the bytes of results are the results of the remote procedure.

8. Network Administration

This chapter is addressed to system and network administrators. It describes the procedures for configuring the network communications software on IRIS-4D Series workstations. Specifically, this chapter covers:

- configuring a new system
- assigning names and addresses and host-address resolution
- network initialization and routing
- network servers
- network databases
- remote access and security
- network troubleshooting
- kernel configuration options

Setting up networked mail is covered in Appendix A.

8.1 Configuring a New IRIS

Here's a summary of the procedure to attach a new IRIS-4D to an existing network. It assumes a basic set-up using */etc/hosts* and not one using BIND or Yellow Pages.

- Choose a name for the system. Insert it in */etc/sys_id.*
- If you're connecting to an existing network, obtain an Internet address from the network administrator. For a new network, choose an address. Update the */etc/hosts* database on the IRIS and on the "master" system for the network to include the address of the IRIS.

- Add the master system's Internet address to the new machine's */etc/hosts* and copy the updated version from the master:

```
rcp guest@master:/etc/hosts /etc
```

  Update the */etc/hosts* file on all other systems on the network so they can access the new host.

- Enable or disable various network daemons with *chkconfig*(1M).
- Customize the network interface information, if necessary.
- Reboot the IRIS.

8.2 Names and Addresses

The first step in configuring your IRIS-4D is to select a unique host name and Internet network address. You must add this name and address pair to the hosts database on your workstation and on all other hosts on your network. The network address is a number that the software uses to identify a machine.

The name of an IRIS workstation is stored in the file */etc/sys_id*. On a new workstation, this file contains the name *IRIS*. The host name can be up to 64 alphanumeric characters long and may include periods and hyphens. Periods are not part of the name but serve to separate components of a domain-style name. Case does not matter; the software converts upper-case letters to lower-case when it translates a host name into an address. Note that after changing */etc/sys_id*, the IRIS's notion of its name doesn't change until the system is rebooted.

Host names are really domain names, where a domain is a hierarchical, dot-separated list of subdomains. For example, the name for the host monet in the Berkeley subdomain of the EDU subdomain of the Internet would be represented as "monet.Berkeley.EDU" (with no trailing dot). The complete name is called a "fully-qualified domain name." If your IRIS is connected to the Internet, use the fully-qualified name in */etc/sys_id*. (It's possible to specify a nickname in the hosts database to avoid typing a long hostname.)

If you are connecting a new workstation to an existing network, obtain an address from the network administrator.

If you are creating a new network, you must generate a series of addresses for your workstations, as explained in the following section.

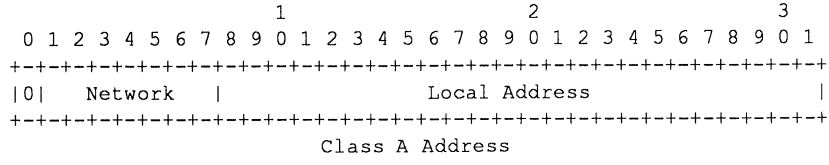
If your organization's networks might be connected to the Internet in the future, contact the Network Information Center at the address listed in Chapter 1 and request a network address assignment. Otherwise, you can choose any series of address you want, based on the information below.

8.2.1 Choosing an Address

The networking software uses a unique number called the Internet address to identify each host. The Internet address is a 4-byte number composed of two logical parts: a network number and a host or local number. The network number is in the most-significant portion of the address and the host number is in the least-significant portion. An address is usually written in "dotted decimal" notation. Each byte is represented as a decimal number between 0 and 255 and separated by a period (for example, 192.77.150.1). See *inet*(3N) for more information on the dot notation.

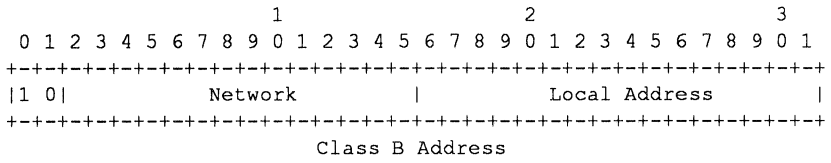
According to "Internet Numbers" (RFC-1117), there are three different classes of Internet addresses for hosts: A, B, and C. An address class tells how to divide the 32-bit address into the network and host numbers. All hosts on a particular network that want to communicate with each other must use the same class of Internet addresses. The most significant or leftmost bits of the address are used to determine the address class.

The first type of address, class A, has a 7-bit network number and a 24-bit local address. The highest-order bit is set to 0. This allows 128 class-A networks, though the networks 0 and 127 are reserved. Each network can have more than 16 million hosts.

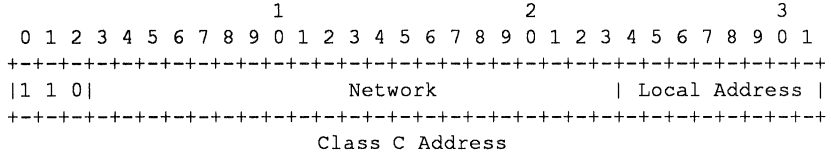


Class A Address

The second type of address, class B, has a 14-bit network number and a 16-bit local address. The two highest-order bits are set to 1-0. In dot notation, network numbers range from 128 to 191 in the first byte. A class-B network can have up to 65534 hosts.



The third type of address, class C, has a 21-bit network number and a 8-bit local address. The three highest-order bits are set to 1-1-0. In dot notation, network numbers range from 192 to 223 in the first byte. A class-C network can have up to 254 hosts.



A fourth type of address, class D, is used as a multicast group address. The four highest-order bits are set to 1-1-1-0. In dot notation, network numbers range from 224 to 239 in the first byte.

No addresses are allowed when the four highest-order bits are set to 1-1-1-1. These addresses, called class E, are reserved.

8.2.2 Special Internet Addresses

Several Internet Protocol (IP) addresses are treated specially by the network software. The class-A network number of 127 is reserved as the loopback network and the address 127.0.0.1 is used by IRIX as the loopback address. Any Internet packet sent to this address is looped back inside the kernel and never reaches the network cable. The loopback address is useful for testing basic functionality of the network software and should never be changed.

The network gives special treatment to two other kinds of Internet addresses: Internet addresses with network numbers that are all zero (0 for class A, 0.0 for class B, and 0.0.0 for class C) or with host numbers that are all ones (255.255.255 for class A, 255.255 for class B, and 255 for class C). An address with a network number of 0 refers to the local network. An address with a host number of all ones is called the broadcast address and refers to all hosts on that network.

The IRIS workstation is shipped with the special class C test address 192.0.2.1 in */etc/hosts*. Change this number to a different one if you plan to connect the IRIS to a network. The network startup script */etc/init.d/network* uses the test address to determine whether or not to go into "standalone network mode" (see below for details). If you do not plan to connect your IRIS to a network, do not change the 192.0.2.1 address.

8.2.3 The Hosts Database

The hosts name-address database on IRIX contains mappings between the Internet addresses and the names for the systems on the network. This information must exist on each system on a network. When you refer to a host name in an application program, the program accesses this database with the *gethostbyname*(3N) routine to find the Internet address of the host.

IRIX has three methods of accessing and maintaining the host name-address database:

- the hosts file, */etc/hosts*,
- the Yellow Pages (YP) service and
- the Berkeley Internet Name Domain service ("BIND name server").

Maintaining consistent versions of */etc/hosts* on every system in a large network is troublesome. Yellow Pages and the BIND name server are other techniques for maintaining a centralized version of the host database. How to choose which service to run is described in the next section.

/etc/hosts

The */etc/hosts* file is an ASCII file that you can modify with any text editor or with the *vadmin*(1G) networking tool. It contains lines of text that specify a host's address followed by its "official" name and any nicknames. The "official" name should be the fully-qualified domain name. The address and name(s) are separated by blanks and/or tabs. Comments begin with a pound sign (#) and continue to the end of the line.

Here is an example of an */etc/hosts* file:

```
# This is a comment
127.0.0.1    localhost
119.0.3.20   mint.spices.com     mint  # mint is a nickname
119.0.3.21   ginger.spices.com   ginger
119.0.3.22   sassafras.spices.com  sassafras sas
```

Each IRIS must have a copy of */etc/hosts* that contains entries for "localhost" and all of its network interfaces. As shipped, the */etc/hosts* file contains two entries. The first entry,

```
127.0.0.1        localhost
```

is a name you can use to test the network software. When you reference *localhost*, the message is looped back internally; it is never transmitted across the network.

Note: Many important programs, such as the 4Sight window system, depend on the "localhost" entry—**do NOT remove or modify it.**

The second entry is the default address and name for your IRIS. To enable the IRIS to access the network, add a new entry that contains a newly-assigned IP address and the name in */etc/sys_id*. The entry **must** contain the sys_id name, either as the official host name or as a nickname. Also, if your IRIS is a gateway, each network interface must be assigned a network address and have an entry in */etc/hosts*, as described in the "Gateways" section below.

Using the example hosts file above, the */etc/sys_id* file for the host ginger should contain either "ginger" or "ginger.spices.com".

If you change the IRIS's name in */etc/sys_id*, make sure to update the entry in */etc/hosts*, otherwise the network software will not initialize properly. If the following message appears during system startup,

```
*** Can't find hostname's Internet address in /etc/hosts
```

then the */etc/hosts* and */etc/sys_id* files are inconsistent and must be fixed.

If you are connecting the IRIS to an existing network, use the */etc/hosts* file on the master system. Make sure it conforms to the syntax specifications summarized above. If you are building a new network, add the name and address for the new workstation as well as for any other workstations on the network to */etc/hosts*.

If the master copy of */etc/hosts* is maintained on a non-Silicon Graphics machine, or you are using BIND or YP, make sure that the hosts database contains the "localhost" entry.

8.2.4 Host-Address Resolution Order

It is important that each machine have a consistent version of the hosts database. Choosing the proper method to maintain the consistency depends on the size of your network and whether the network is connected to the Internet.

For a small network of machines under the same administrative control, maintaining a consistent */etc/hosts* file is straightforward. Designate one machine as having the master copy and make additions or deletions from its file, then use *rcp*(1C) or *rdist*(1C) to copy the file to the other machines in the network.

For moderately-sized networks (roughly more than 20 hosts or a small collection of interconnected networks), the Yellow Pages distributed lookup service is a good choice. YP is part of the NFS optional software and is described in the *NFS User's Guide.*

For large networks or ones connected directly or indirectly to the Internet, use BIND for your host name and address mapping. BIND provides access to a much larger set of hosts than are provided in the *hosts* file. A drawback of BIND is its complicated setup. BIND is described in more detail in the chapter entitled "The BIND Name Server."

When you use BIND or Yellow Pages, the file */etc/hosts* is only used for setting interface addresses and by *netstat*(1M). The file is also used when these services are not running. Therefore, it need only contain addresses for all of your IRIS's network interfaces and a few important hosts on the network.

Most IRIX commands try to access the YP database first, then BIND and finally the *hosts* file. If YP or BIND are running and don't have the requested information in their database, the other services are not queried. For sites connected to the Internet that want to use YP for everything but the hosts database, this behavior is a problem. There are two ways to fix this. The first method is to run Yellow Pages and BIND at the same time, using YP for the local hosts and BIND for the Internet hosts.

The Yellow Pages servers for a YP domain can access the name server to obtain interdomain host information. On each server, create the file */etc/config/ypserv.options* and add the string "–i" as follows:

```
echo "-i" > /etc/config/ypserv.options
```

The YP servers must also be configured as BIND clients, as described in the chapter about BIND.

The second method is to bypass YP altogether. The system administrator can change the default host resolution ordering by adding a "hostresorder" line to */usr/etc/resolv.conf* (this keyword is described in *resolver*(4)). For example, the line

```
hostresorder bind local
```

specifies that *gethostbyname*(3N) and *gethostbyaddr*(3N) access the BIND name server first and if no information was found, to access */etc/hosts*, bypassing the YP hosts database. When using BIND, the other lines in the *resolv.conf* file must be set up correctly in order to access the name server. See the "The BIND Name Server" chapter and *resolver*(4) for more details.

The *hostresorder* mechanism is useful when accessing an unreliable remote BIND name server or one over a slow or distant network or one that does not have the appropriate host entries for your machine. With the following

```
hostresorder local bind
```

to check */etc/hosts* first before accessing the name server, you can increase reliability of name service at the cost of possibly using out-of-date information for remote hosts.

8.3 Network Initialization

During system initialization, the shell script */etc/init.d/network* is run. This script configures the network devices and starts the appropriate daemon processes. During system shutdown, the script kills the daemons and disables the network devices. The following subsections describe the actions of this script in more detail.

The script */etc/init.d/network* is linked to */etc/rc0.d/K40network*, which is invoked from */etc/rc0* during shutdown, and to */etc/rc2.d/S30network*, which

is invoked from */etc/rc2* during startup. The script understands two arguments: *start* and *stop*. When invoked with *start*, the script sets the IRIS's host name and ID using the */etc/sys_id* and */etc/hosts* files; the host ID is also the system's Internet address in */etc/hosts*.

If the host ID is the default test address, 192.0.2.1, then the network software is initialized for 'standalone' operation. Standalone mode is equivalent to unplugging the network cable from the machine: outgoing network traffic is not generated and incoming traffic is not received. Only required daemons are started. Most configurable daemons are not started because they are needed for remote operations.

During initialization, the network script uses configuration flag files in the */etc/config* directory to determine whether to start each daemon. The *chkconfig*(1M) command is used to check (and change) the state of each flag. If the flag file exists and is in "on" state, the daemon is started.

The daemon is not started if the flag file does not exist or is off. Some daemons take optional or required arguments, which should be placed in 'options' files in */etc/config*.

Two configuration flags in */etc/config* control the initialization process. If the flag *verbose* is on, the script will print informative messages on the console as daemons are started and devices configured. If the flag *network* is off, the script will initialize the network software for the standalone mode. To start and terminate locally-developed network daemons, or to publish ARP entries and load routes, create the separate shell script */etc/init.d/network.local*. Make symbolic links in */etc/rc0.d* and */etc/rc2.d* to this file to have it called during system startup and shutdown:

```
ln -s /etc/init.d/network.local /etc/rc0.d/K39network
ln -s /etc/init.d/network.local /etc/rc2.d/S31network
```

See */etc/init.d/network* for the basic format of the script. Also refer to *chkconfig*(1M) and *network*(1M) for more information.

8.3.1 Gateways

If your IRIS-4D has more than one network interface, it can act as as a gateway (or packet forwarder) between the networks. If the network startup script discovers a second ethernet interface, it is automatically configured. A HyperNET interface is configured if the configuration flag *hypernet* is

"on" (see *chkconfig*(1M) for details). Each interface must have an unique Internet address and name entered in */etc/hosts*. The script derives the names from the IRIS' name in */etc/sys_id*. The prefix *gate-* is prepended to the host name to generate the second interface's name and for HyperNet, the suffix *-hy* is appended to generate that interface's name.

For example:

```
191.50.1.7      yosemite
191.50.2.49     gate-yosemite
191.50.3.8      yosemite-hy
```

If you have more than 2 ethernet interfaces, edit */etc/init.d/network* and follow the instructions in the script for enabling the additional boards.

Several kernel configuration options affect gateway machines. See the section titled "Kernel Configuration Options" for details.

8.3.2 Device Configuration

The network interface drivers for ethernet and HyperNET controllers require that their Internet addresses be defined at system startup. This is done with the *ifconfig*(1M) command. Each interface must have an entry in */etc/hosts*. Note that *ifconfig* uses the hosts file only, even if BIND or YP are running.

You can also use the *ifconfig* command to set options for the interface at boot time. These options include setting the network mask and broadcast address and disabling the use of the Address Resolution Protocol (ARP). Options are set independently for each interface, and apply to all packets sent using that interface. Options for each interface should be put into files in */etc/config*: *ifconfig-1.options* for the primary interface, *ifconfig-2.options* for the second interface, and *ifconfig-hy.options* for the HyperNET interface, etc.

For example, to create *ifconfig* options for the primary interface, type:

```
echo "netmask 0xFFFFFF00" > /etc/config/ifconfig-1.options
```

Frequently used *ifconfig* options are described in the next sections.

8.3.3 Local Subnetworks

Subnetworks allow multiple local networks to appear as a single Internet network to off-site hosts. Subnetworks are useful because they allow a site to hide its local topology, requiring only a single route in external gateways. Local network numbers can be locally administered. The standard describing this change in Internet addressing is RFC-950.

To set up local subnetworks, first decide how to partition the host part of the 32-bit Internet address. To define local subnets, you must steal bits from the host number sequence for use in extending the network portion of the Internet address. This reinterpretation of Internet addresses is done only for local networks. It is not visible to hosts off-site.

Sites with a class-A network number have 24 bits of host numbers with which to work. Sites with a class-B network number have 16 bits of host numbers. Sites with a class-C network number have 8 bits of host numbers. For example, if your site has a class-B network number, hosts on this network have an Internet address that contains 16 bits for the network number and 16 bits for the host number. To define 254 local subnets, each possessing at most 254 hosts, 8 bits may be taken from the host part. These new network numbers are then constructed by concatenating the original 16-bit network number with the extra 8 bits containing the local subnetwork number. At least 2 bits should be used for subnetwork numbers.

Note: The use of subnets 0 and all-1s is discouraged to avoid confusion with broadcast addresses.

The existence of local subnetworks is communicated to the system when the network interface is configured with the *netmask* option to the *ifconfig* command. A network mask defines the portion of the Internet address that is to be considered the network part. This mask normally contains the bits corresponding to the standard network part as well as the portion of the host part that has been assigned to subnets. If no mask is specified when the address is set, it will be set according to the class of the network. For example, if the primary network interface used a class B network such as 128.32 and used 8 bits of the host part for defining subnetworks, the */etc/config/ifconfig-1.options* file would contain:

```
netmask 0xffffff00
```

This specifies that for that interface, the upper 24 bits of the Internet address should be used in calculating network numbers (netmask 0xffffff00).

Host *m* on sub-network *n* of this network would have addresses of the form:

```
128.32.n.m
```

For example, host 99 on subnetwork 129 would have this address:

```
128.32.129.99
```

For hosts with multiple interfaces, the network mask should be set for each interface in the appropriate *ifconfig* options files.

8.3.4 Internet Broadcast Addresses

The broadcast address for Internet networks (according to RFC-1117) is indicated by a host part of all 1's. The address used by older, 4.2BSD-derived systems was the address with a host part of 0. IRIX uses the standard broadcast address (all 1's) by default, but allows the broadcast address to be set (with *ifconfig*) for each interface by including it in the appropriate *ifconfig* options files. This allows networks consisting of both 4.2BSD and IRIX hosts to coexist. For example, if the interface's address is 192.0.2.34, the following argument to *ifconfig* sets the broadcast address to the old style:

```
broadcast 192.0.2.0
```

In the presence of subnets, the broadcast address uses the subnet field, as do normal host addresses, with the remaining host part set to 1's (or 0's, on a network that has not yet been converted). IRIX hosts recognize and accept packets sent to the logical-network broadcast address, as well as those sent to the subnet broadcast address. Also, when using an all-1's broadcast, IRIX hosts recognize and receive packets sent to host 0 as a broadcast.

8.3.5 Publishing ARP Entries

To send packets on an ethernet to other hosts, a host must be able to translate the Internet addresses into ethernet hardware addresses. On IRIX and 4.3BSD systems, this translation is done dynamically with the Address Resolution Protocol (ARP). Network software also supports communication with hosts that do not use ARP. An IRIX host can use the *arp*(1M) command to set (or "publish") translations for non-ARP hosts in advance.

For example, the system named ginger does not support ARP and has the ethernet address 8:0:20:1:74:89. To add an arp entry to the translation table for ginger, as *root*, type:

```
arp -s ginger 8:0:20:1:74:89  pub
```

The –s option tells *arp* to add the entry to the translation table. The **pub** option "publishes" the entry; it enables your IRIS to respond to an ARP broadcast for the other host's address.

Once you add an entry to the table, it remains there until you reboot. If you want to make the entry permanent, create */etc/init.d/network.local* (see the beginning of this section to create this file) and add the appropriate *arp* commands. Alternately, put multiple ARP entries in a file, and to publish them, use:

```
arp -f filename
```

8.3.6 Routing

To access networks not directly attached to your host, your system must obtain information to allow packets to be properly routed. Two schemes are supported. The first (and default) scheme uses the routing table management daemon *routed*(1M). This daemon uses a variant of the Xerox Routing Information Protocol (RIP) to automatically maintain up-to-date routing tables in a cluster of local area networks. By using */etc/gateways*, the routing daemon can also be used to initialize static routes to distant networks. When the routing daemon is started during system initialization, it reads */etc/gateways* (if it exists) and installs the routes defined there. It then broadcasts on each attached local network to find other instances of the routing daemon. If any responses are received, the routing daemons cooperate to maintain a globally consistent view of routing in the local environment. You can extend this view to include remote sites also running the routing daemon by setting up suitable entries in */etc/gateways*; consult *routed*(1M) for details.

The *gated*(1M) routing daemon handles the HELLO and EGP routing protocols in addition to RIP. Use *gated* if your IRIS acts as a gateway to an external network that uses EGP or HELLO. See the *gated* manual page for information on setting up the gated configuration file.

The second approach is to define a default or wildcard route to a smart gateway and depend on the gateway to provide ICMP routing redirect information to create dynamically a routing data base. This is done by adding an entry to */etc/init.d/network* after *routed* is started. The entry has this syntax:

```
/usr/etc/route add default smart-gateway 1
```

See *route*(1M) for more information. The default route will be used by the system as a last resort in routing packets to their destination. Assuming the gateway to which packets are directed is able to generate the proper routing redirect messages, the system will then add routing table entries based on the information supplied. This approach is unsuitable in an environment where there are only bridges (i.e., pseudo gateways that do not generate routing redirect messages). Furthermore, if the smart gateway goes down, there is no alternative but to create routes manually with *route*(1M) to maintain service.

The system always listens for, and processes, routing redirect information, so you can combine both of the above facilities. For example, you could use the routing table management process to maintain up-to-date information about routes to geographically local networks, while employing the wildcard routing techniques for distant networks.

You can use the *netstat*(1M) program to display routing table contents as well as various routing oriented statistics. For example, this command:

```
/usr/etc/netstat -r
```

will display the contents of the routing tables, while:

```
/usr/etc/netstat -r -s
```

will show the number of routing table entries dynamically created as a result of routing redirect messages.

8.3.7 Multicast Routing

The network startup script, */etc/init/network* sets the default route for all IP multicasts to the primary network interface. Selection of the default multicast interface is controlled via the kernel (unicast) routing table. If there is no multicast route in the table, all multicasts will, by default, be sent on the interface associated with the default gateway. If that interface does not support multicast, attempts to send will receive an ENETUNREACH error.

A route may be added for a particular multicast address or for all multicast addresses, to direct them to a different default interface. For example, to specify that multicast datagrams addressed to 224.0.1.3 should, by default, be sent on the primary interface on host yosemite, use the following:

```
/usr/etc/route add 224.0.1.3 yosemite 0
```

To change the default for all multicast addresses, other than those with individual routes, to be the secondary interface on host yosemite, use:

```
/usr/etc/route add 224.0.0.0 gate-yosemite 0
```

If you point a multicast route at an interface that does not support multicasting, an attempt to multicast via that route will receive an ENETUNREACH error.

If needed, you can insert multicast routes in */etc/gateways* or add the appropriate commands to the */etc/init.d/network.local* file, so they take effect every time the system is booted.

8.4 Network Servers

Important network servers (or *daemons*) such as *routed* and *inetd* are automatically started up at boot time by the network startup script. Other servers are started by the Internet daemon, *inetd*(1M). The *sendmail* daemon is started from */etc/init.d/mail.*

The following daemons are started if their configuration flag is on. If optional software packages (such as NFS, Yellow Pages, and 4DDN) are not installed, the script will not try to start them. Several daemons use or

require *.options* files to modify their behavior. Consult the */etc/init.d/network* script and the manual pages for details about their operation.

Config. Flag	Function	*.options* File?
gated	multiprotocol routing daemon	yes
mrouted	IP multicast routing daemon	yes
named	BIND name server	yes
timed	clock synchronizer	yes
timeslave	clock synchronizer	required
rwhod	system status daemon	yes
nfs	NFS remote file systems	no
automount	NFS automounter daemon	yes
rarpd	Reverse ARP daemon	yes
yp	Yellow Pages distr. lookup	no
ypserv	Become YP server	yes
ypmaster	Become YP password master	yes
4DDN	Enable 4DDN software	no

Table 8-1. Network Daemons and Their Function

8.4.1 Inetd

In IRIX, most of the Interner server programs are started up by a "super server" daemon called *inetd*. The daemon acts as a master server for programs specified in its configuration file, */usr/etc/inetd.conf*. It listens for service requests for these servers, and starts up the appropriate program whenever a request is received. Lines in the configuration file specify the following:

- the service name. The three types of services are: Internet (as found in */etc/services* or in the Yellow Pages *services* database), rpc (in */etc/rpc* or the YP *rpc.bynumber* database) and tcpmux.
- the type of socket the server expects (e.g., stream or dgram)
- the protocol to be used with the socket (as found in */etc/protocols* or in the Yellow Pages *protocols* database)
- whether to wait for each server to complete before starting up another
- the user name used by the server when it runs

- the server program's name
- up to 11 arguments to pass to the server program. The first argument must be the program name.

For example, an entry for the file transfer protocol server would appear as:

```
ftp    stream  tcp    nowait    root    /usr/etc/ftpd    ftpd
```

Some trivial services are implemented internally in *inetd*, and their server program names are listed as "internal."

To have local network servers started from *inetd*, the appropriate lines should be added to the configuration file */usr/etc/inetd.conf*. Consult *inetd*(1M) and the "Network Programming" chapter for more detail on the format of the configuration file and the operation of the Internet daemon.

8.5 Network Databases

Several data files are used by the network library routines and utilities:

File	Manual reference	Use
/etc/hosts	*hosts* (4)	host names
/etc/hosts.equiv	*hosts.equiv* (4)	list of "trusted" hosts
/etc/hosts.lpd	*lpd* (1M)	hosts allowed to access printers
/etc/ftpusers	*ftpd* (1M)	list of "unwelcome" ftp users
/etc/networks	*networks* (4)	network names
/etc/protocols	*protocols* (4)	protocol names
/etc/rpc	*rpc* (4)	rpc program numbers
/etc/services	*services* (4)	list of known Internet services
/usr/etc/inetd.conf	*inetd* (1M)	list of servers started by *inetd*
/usr/etc/resolv.conf	*resolver* (4)	host name lookup

Table 8-2. Network Data Files

The distributed files are set up for a minimal configuration. You must be logged in as *root* in order to edit these files.

The */etc/hosts.equiv* file contains a list of trusted machines and is described in more detail later in this chapter. */etc/ftpusers* contains a list of prohibited and restricted ftp users and is described in detail later in this chapter.

The */etc/hosts* file, which was described above, is the host name-address database. The */etc/networks* file contains the network name-address database. Each network in your local-area network should be added to this file. (*netstat*(1M) uses this file when printing routes.) The */etc/protocols*, */etc/rpc*, and */etc/services* usually do not need to be changed. They contain mnemonic names for protocols, RPC protocol numbers and Internet services.

For small networks, use *rdist*(1) to maintain consistent versions of theses files. Larger networks may want to use the Yellow Pages (part of the NFS optional software) for the following files: hosts, networks, protocols, services.

8.6 Remote Access and Security

The remote login and shell servers use an authentication scheme based on "trusted hosts." The */etc/hosts.equiv* file contains a list of hosts that are considered trusted and under the same administrative control. When a user contacts a remote login or shell server requesting service, the client process passes the user's name and the official name of the host on which the client is located. If the host's name is located in *hosts.equiv* and the user has an account on the server's machine, then service is rendered (i.e., the user is allowed to log in, or the command is executed). Users may expand this "equivalence" of machines by installing a *.rhosts* file in their login directory. The *root* login bypasses the */etc/hosts.equiv* file, and uses only *root*'s (typically */.rhosts*) file.

Create *root*'s */.rhosts* only if all systems and their consoles are physically secure and all privileged accounts have passwords. Be selective about the systems you add to the file. Given access to a console on a machine with */.rhosts* privileges, someone can log in as any user, including the superuser, and become *root* on any system that has your system's name and *root* in its */.rhosts* file.

To create a class of equivalent machines, include the *official* names for those machines in the */etc/hosts.equiv* file. If you are running the name server, you may omit the domain part of the host name for machines in your local domain.

For example, if the following machines on a network are considered trusted, the */etc/hosts.equiv* file lists them as follows:

```
lassen
redwood
sequoia
yosemite
```

The *hosts.equiv* file can have a line of the form

```
host user
```

with white space separating the names. This allows the specified user on the remote machine to log in as anyone!

The owner of the *.rhosts* file must be the super-user (i.e., *root*) or the user in whose home directory it resides. The */etc/hosts.equiv* file must be owned by *root*. The contents of the files will be disregarded if they are owned by another user or if their permissions allow anyone who is not the owner to modify the file. Use the *chmod* command to add the proper protection:

```
chmod  go-w  .rhosts
```

It's important that all accounts on machines connected to Internet have passwords. Use *passwd*(1) to give an account a valid or locked password.

Remote shell accesses and remote logins are disabled if */etc/nologin* exists. For remote logins, the contents of the file are printed before the connection is closed. This is useful, for example, to notify users of expected system availability after a disk backup. Some sites require that a notice be printed before remote logins and ftp sessions. Create and edit the file */etc/issue* to contain such a message.

8.6.1 Remote Access Logging

Several network daemons have an option to log remote accesses to the system log file */usr/adm/SYSLOG* using *syslogd*(1M). Sites connected to the Internet should use this feature. To enable logging for *ftpd*(1M), *tftpd*(1M), and *rshd*(1M), edit */usr/etc/inetd.conf* and add –l after the right-most instance of ftpd and tftpd, and add –L after the right-most instance of rshd. Signal inetd to reread its file.

```
/etc/killall -HUP inetd
```

Remote logins via *rlogin*(1), *telnet*(1), and the 4DDN *sethost*(1) programs can be logged by *login*(1). Create */etc/config/login.options* and add the keyword "syslog=all" or "syslog=fail" to it. For example,

```
echo "syslog=all" > /etc/config/login.options
```

will log successful and failed local and remote login attempts to *syslogd*(1M). See the *login*(1) manual page for details.

8.6.2 Anonymous and Restricted FTP Access

The FTP server included in the system provides support for restricted FTP access, including an anonymous account. Because of the inherent security problems with such a facility, read this section carefully if you consider providing such a service.

Aside from the problems of publicly writable directories, the ftp server may provide a loophole for interlopers if certain user accounts are allowed. The file */etc/ftpusers* is checked on each connection. If the requested user name is located in the file, the request for service is denied (excepted as noted below). Accounts with nonstandard shells should be listed in this file. Accounts without passwords need not be listed in this file; the ftp server will not service these users.

A restricted account has limited access to files on the system. When a client uses the restricted account, the server performs a *chroot*(2) system call to restrict the client from moving outside that part of the file system where the account's home directory is located. Because the server uses a *chroot* call, certain programs and files used by the server process must be placed in the account's home directory. Furthermore, be sure that all directories and executable images are unwritable.

A restricted account must be listed in */etc/ftpusers* with a line containing the account name followed by the word "restricted". For example,

```
support restricted
```

specifies the "support" account allows restricted ftp access. A client must specify a password to access a restricted account. When the account logs in, the file called README in the account's home directory is printed, if it exists, before the client can execute commands. The README file is a good place for system notices and account usage policy.

An "anonymous" account is special type of restricted account that does not require a password and does not need to be listed in the ftpusers file. You can enable an anonymous account by creating an entry in */etc/passwd* for the user *ftp*:

```
ftp:*:997:999:FTP anonymous account:/usr/people/ftp:/dev/null
```

The "*" for the password prevents anyone from logging into the account by any other means. Restricted accounts also need an entry in */etc/passwd.* The system manager should set the account's password using the *passwd*(1) command.

Here is a recommended directory setup for any restricted account — the anonymous account, ftp, is used as an example.

```
cd ~ftp
chmod 555 .
chown ftp .
chgrp ftp .
mkdir bin etc pub
chown root bin etc
chmod 555 bin etc
chown ftp pub
chmod 777 pub    # allows anyone to add/delete files
cd bin
cp /bin/ls .
chmod 111 ls
cd ../etc
cp /etc/passwd /etc/group .
chmod 444 passwd group
```

To place files in the restricted/anonymous area, local users must place the files in a subdirectory. In the configuration described above, they are placed in the *pub* subdirectory.

An important issue to consider is the *passwd* file in the restricted account's directory. This file can be copied by users who use the account. They may then try to break the passwords of users on your machine for further access; therefore, remove any unnecessary users and change all passwords in this file to an invalid password such as "*". A good choice of users to include in this copy of the */etc/passwd* file might be root, bin, sys, and the restricted account.

8.7 Network Troubleshooting

If you experience difficulty with network connections, first check the physical network connections to the problem machine. Check the cable, transceiver, and tap. A loose or detached cable is a common cause of the difficulty. Make sure the cable is securely connected to the machine.

If the cable is attached properly, use the *ping*(1M) command to determine if your machine can communicate with other machines on the network. For example, if network access to the host ginger fails, try to determine if ginger is still running with

```
ping ginger
```

If ginger doesn't respond to *ping*, make sure you can *ping* other hosts on your local network. (You can use a host's IP address instead of its name.)

If the *ping* to ginger reponds but the *rsh*(1), *rlogin*(1), or *telnet*(1) commands to access ginger do not respond, make sure the *inetd* server running is running on ginger.

If the remote host is on a different network, make sure there is a route to it. Communication to the host will be impossible if the gateway(s) between your network and the other host's is down. Use the **–r** option to the *netstat*(1M) command to diagnose routing problems in networks with gateways.

The *netstat* program can also help track down hardware malfunctions and network load. In particular, the **–i** option shows the number of packets received and transmitted for each configured interface. This information is useful to determine if the network is saturated. If the ratio of collisions to total traffic is significant, say more than 20%, the network should be divided into subnets. The **–s** option shows statistics for the IP, ICMP, IGMP, TCP, and UDP protocols. The **–m** option shows memory usage.

See the *ethernet*(7) manual page for information about error messages from the IRIS-4D ethernet controllers.

Additional network management and troubleshooting programs are available with the NetVisualyzer™ optional software.

8.8 Kernel Configuration Options

You can change several parameters to customize network behavior for local configurations. The parameters listed in this section are in the */usr/sysgen/master.d/bsd* configuration file. To learn how to configure the kernel after changing this file, see the *System Tuning and Configuration Guide* for details.

ipgateway

The machine is to be used as a gateway. Machines that have only a single hardware network interface will not forward IP packets; when this option is off, they will also refrain from sending any error indication to the source of unforwardable packets. Gateways with only a single interface are assumed to have missing or broken interfaces, and will return ICMP unreachable errors to hosts sending them packets to be forwarded. (Default value = 0 "off")

ipforwarding

Normally, IRIX machines with multiple network interfaces will forward IP packets that should be resent to another host. If ipforwarding is set to 0, IP packet forwarding will be disabled. (Default = 1 "on")

ipsendredirects

If ipforwarding is turned on and if a packet is forwarded back through the same interface on which it arrived, IRIX will send an ICMP redirect to the source host; this works only if the source host is on the same network. This ability to redirect packets improves the interaction of IRIX gateways with hosts that configure their routes via default gateways and redirects. If the packet was forwarded using a route to a host or to a subnet, a host redirect is sent, otherwise a network redirect is sent. You can disable the generation of redirects by changing the value of ipsendredirects to 0 in environments where it may cause difficulties. (Default = 1 "on")

subnetsarelocal
allnetsarelocal

TCP calculates a maximum segment size to use for each connection, and sends no datagrams larger than that size. This size will be no larger than that supported on the outgoing interface. If the destination is not on the local network, the size will be no larger than 576 bytes. It takes far longer to send data using 576-byte packets than it does

using 1500-byte packets. If the subnetsarelocal parameter is set to 1, other subnets of a directly-connected network are considered to be local. If the allnetsarelocal parameter is set to 1, other networks are considered to be local. This is useful for sites with a connected set of class C networks not connected to an external network such as Internet. (subnetsarelocal default = 1 "on", allnetsarelocal default = 0 "off")

ipcksum
tcpcksum
udpcksum

These parameters affect whether IP headers, TCP headers and data, and UDP headers and data are checksummed. By default, the checksums are calculated. It is strongly recommended that you do not disable them. However, if you must use UDP with 4.2BSD hosts, set the udpcksum flag to 0.

tcp_sendspace
tcp_recvspace
udp_sendspace
udp_recvgrams

These parameters determine the default amount of buffer space used by a TCP (SOCK_STREAM) and UDP (SOCK_DGRAM) sockets. The tcp_sendspace and tcp_recvspace parameters define the initial buffer space allocated to a socket. The udp_sendspace parameter defines the default maximum size of UDP datagrams that can be sent. The udp_recvgrams parameter determines the number of maximally-sized UDP datagrams that can be buffered in a UDP socket. The total receive buffer size in bytes for each UDP socket is the product of udp_sendspace and udp_recvgrams. A program can increase or decrease the send and receive buffer sizes for a socket with the SO_SNDBUF and SO_RCVBUF options to the *setsockopt*(2) system call.

For 4.2BSD compatibility, the IRIX system limits its initial TCP sequence numbers to positive numbers.

9. The BIND Name Server

The Berkeley Internet Name Domain (BIND) server implements the Internet Domain Name System (DNS) for the IRIX operating system. A name server is a network service that enables clients to name resources or objects and share this information with other objects in the network. This in effect is a distributed database system for objects in a computer network. BIND is fully intergrated into IRIX network programs for use in storing and retrieving host names and addresses. You can configure the system to use BIND to replace the original host table lookup of information in the network hosts file */etc/hosts*.

BIND has two parts: the name server program, *named*, and a set of C library "resolver" routines to access the server. *named* is a daemon that runs in the background and responds to UDP and TCP queries on a well-known network port. The library routines reside in the standard C library, *libc.a*. The host-address lookup routines *gethostbyname* (3N), *gethostbyaddr* (3N), and *sethostent* (3N) calls use the resolver routines to query the name servers. The resolver library routines described in *resolver*(3N) include routines that build query packets and exchange them with the name server.

This chapter describes how to configure your system to use the BIND name server, and includes an explanation of:

- BIND servers and clients
- how to set up your domain
- standard resource record format
- domain management

9.1 The Domain Name Service

The basic function of the name server is to provide information about network objects by answering queries. The specifications for this name server are defined in RFCs 974, 1034, and 1035. You can obtain these documents from the Network Information Center (NIC) at the address listed in Chapter 1 or you can *ftp* them from the RFC: directory on NIC.DDN.MIL using the "anonymous" account. Aslo refer to the related manual pages, *named*(1M), *resolver*(3N), and *resolver*(4) for additional details.

The advantage of the name server over host table look-up is that it avoids the need for a single centralized clearinghouse for all names. The authority for this information can be delegated to the different organizations on the network responsible for it.

On the other hand, the host table look-up routines require that the master file for the entire network be maintained at a central location by a few people. This works well for small networks where there are only a few machines and there is cooperation among the different organizations responsible for them. However, this does not work well for large networks where machines cross organizational boundaries.

With the name server, the network can be broken into a hierarchy of domains. The name space is organized as a tree according to organizational or administrative boundaries. Each node in the tree, called a *domain*, is given a label. The name of the domain is the concatenation of all the labels of the domains from the root to the current domain. The labels are listed from right to left and are separated by dots. A label need only be unique within its domain. The whole space is partitioned into several nonoverlapping areas of authority called *zones*. Information in each zone is handled by the zone's "authoritative" or "master" name server(s). Each zone starts at a domain and extends down to the leaf domains, or to domains where other zones start. Zones usually represent administrative boundaries.

The current top-level domains registered with the Network Information Center are:

arpa — a temporary domain for hosts, also used as the top-level domain for address to name mapping

com — companies and businesses

edu	universities and other educational institutions
gov	government agencies
mil	military organizations
net	various network-type organizations, network management-related organizations, such as information centers and operations centers
org	technical-support groups, professional societies, or similar organizations

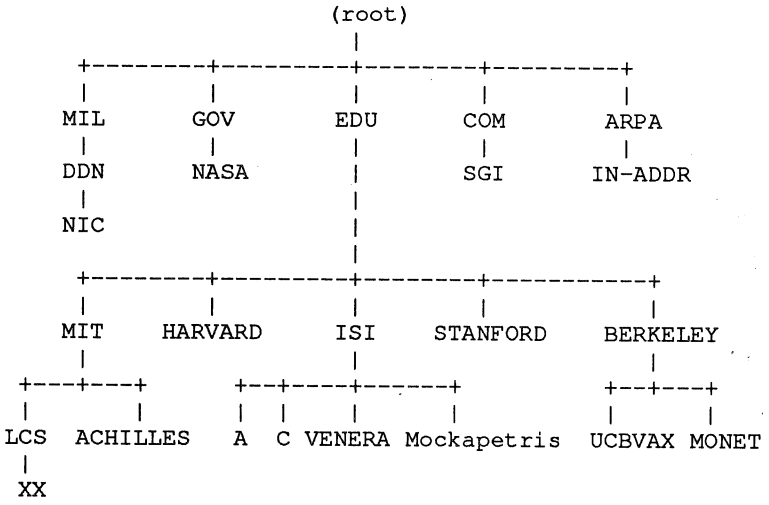
plus a number of top-level country domains.

An example of a domain name for a host at the University of California, Berkeley is:

```
monet.Berkeley.EDU
```

The top-level domain for educational organizations is EDU. Berkeley is a subdomain of EDU, and monet is the name of the host.

The following figure shows a part of the current domain name space. Note that the tree is a very small subset of the actual name space.



In this example, the root domain has five immediate subdomains: MIL, GOV, EDU, COM, and ARPA. The LCS.MIT.EDU domain has one immediate subdomain named XX.LCS.MIT.EDU. All of the leaves are also domains.

9.2 BIND Servers and Clients

You can configure the name server in several ways, depending on the degree of authority and network connectivity.

9.2.1 Master Servers

A master server for a domain is the authority for that domain. This server maintains all the data corresponding to its domain. Each domain should have at least two master servers: a primary master and a secondary master to provide backup service if the primary is unavailable or overloaded. A server can be a master for multiple domains, being primary for some domains and secondary for others.

A primary master server is a server that loads its data from a file on disk. This server can also delegate authority to other servers in its domain. A secondary master server is a server that is delegated authority and receives

its data for a domain from a primary master server. At boot time, the secondary server requests all the data for the given zone from the primary master server. This server then periodically checks with the primary server to see if it needs to update its data.

Root servers are the master servers for the root and top-level Internet domains. They are listed in the *root.cache* file described below.

9.2.2 Caching-Only Server

A caching-only server is not authoritative for any domain. It services queries and asks other servers, who have the authority, for needed information.

Note that all servers cache data until the data expire, based on a time-to-live field attached to the data received from another server.

9.2.3 Slave and Forwarding Servers

A slave server always forwards queries it cannot satisfy locally to a fixed list of forwarding servers, instead of interacting with the master name server for the root and other domains. There may be one or more forwarding servers, and they are tried in turn until the list is exhausted. The queries to the forwarding servers are recursive.

Use a slave-and-forwarder configuration when you do not want all the servers at a given site to interact with the rest of the Internet servers. A typical scenario would involve a number of workstations and a departmental gateway machine with Internet access. The workstations might be administratively prohibited from having Internet access. To give the workstations the appearance of access to the Internet domain system, the workstations could be slave servers to the forwarding server on the gateway machine. The gateway server would forward the queries and interact with other name servers on on the Internet to resolve the query before returning the answer. A benefit of using the forwarding feature is that the central machine develops a more complete cache of information that all the workstations can take advantage of. Using slave mode and forwarding is discussed further in the "Boot File" section of this chapter.

9.2.4 Clients

A BIND client accesses the name servers that run on other machines in the network. The *named* server doesn't run on the local machine.

9.3 Setting Up Your Domain

To set up a domain on a public network, contact the organization in charge of the network and request the appropriate domain registration form. An organization that belongs to multiple networks (such as CSNET and the Internet) should register with only one network.

9.3.1 Internet

Sites that are already on the Internet and need information on setting up a domain should contact the Network Information Center You can reach NIC by electronic mail at *HOSTMASTER@NIC.DDN.MIL* or by telephone (the number is listed in Chapter 1). Obtain a domain questionnaire from the NIC hostmaster, or *ftp* the files NETINFO:DOMAIN-TEMPLATE.TXT and NETINFO:IN-ADDR-TEMPLATE.TXT from the host NIC.DDN.MIL. When registering a domain for a connected network, be sure to register the reverse address domain (IN-ADDR.ARPA) for your networks.

The BIND mailing list is a mail group for people on the Internet running BIND. The group discusses future design decisions, operational problems, and related topics. Send requests to be placed on this mailing list to:

```
bind-request@ucbarpa.Berkeley.EDU.
```

9.3.2 CREN/CSNET

A *CREN/CSNET* member organization that has not registered its domain name should contact the *CSNET* Coordination and Information Center (*CIC*) for an application and information about domain setup. You can reach the *CIC* by electronic mail at *cic@sh.cs.net*, or by phone at (617) 873-2777.

9.4 Files

This section describes how to set up *named*'s database files.

In IRIX, the named database files are stored in the */usr/etc/named.d* directory. A README file contains a short summary of the set-up procedure and a list of official names for IRIS-4D machines. The */usr/etc/named.d/Examples* subdirectory contains sample *named* database files. Based on the following explanation, copy the appropriate example files into */usr/etc/named.d* and edit them to correspond to your system's setup.

You need to set up the following files:

1. *named.hosts*: the host-address database file for your domain; mandatory for primary servers.
2. *named.rev*: the address-host database file for your domain; mandatory for primary servers.
3. *named.boot*: the startup file; mandatory for any type of server.
4. *localhosts.rev*: the "localhost" address-name database file; required on any type of server.
5. *root.cache*: the "cache" of the Internet root servers; required on any type of server.
6. *resolv.conf*: resolver defaults; mandatory for clients, optional for servers.

9.4.1 /usr/etc/resolv.conf and /etc/hosts

The file */usr/etc/resolv.conf* is read the first time *gethostbyname*(3N) or *gethostbyaddr*(3N) is called. It has several functions:

- defines the default domain or the default domain search list
- specifies the ordering of host resolution services used by *gethostbyname*(3N) and *gethostbyaddr*(3N)
- lists Internet addresses of name servers

The first two items apply to both client and server hosts. The last item is required only by client hosts. The file's format is described in detail in *resolver*(4).

To set up a host as a client of remote servers, add "nameserver" entries for the Internet addresses of the name servers to */usr/etc/resolv.conf*. For example:

```
nameserver 128.32.130.12
```

Up to 3 nameserver entries can be specified. It is usually not necessary to create this file if you have a local server running. An entry for the local server should use an Internet address of 0.

On client and server hosts, the hostname in */etc/sys_id* should be set to the fully-qualified domain name. For example:

```
monet.Berkeley.EDU
```

However, if you choose not to do so, add a line with the keyword "domain" and the host's domain to the *resolv.conf* file. For example:

```
domain berkeley.edu
```

The *gethostbyname*(3N) and *gethostbyaddr*(3N) library routines are normally configured to access the Yellow Pages (YP) hosts map if YP is running, else *named* if it is running, or */etc/hosts* to resolve an address. This can be changed with the *hostresorder* keyword in */usr/etc/resolv.conf*. See Chapter 8 and *resolver*(4) for details.

To enable the system manager to *rcp* files from another machine when the system is in single-user mode, the */etc/hosts* file should contain entries for important hosts in addition to the entries for the local machine's network interface(s). See *hosts*(4) for more information about the format.

9.4.2 Boot File

The boot file is first read when *named* starts up. This tells the server what type of server it is, which zones it has authority over, and where to get its initial data. The name of this file is */usr/etc/named.d/named.boot*, the default.

To use a different file, create */etc/config/named.options*:

```
echo "-b other-bootfile-name" > /etc/config/named.options
```

The recognized bootfile commands are described in the next sections.

Directory

The directory line specifies the directory in which the name server should run, allowing the other file names in the boot file to use relative path names.

```
directory        /usr/etc/named.d
```

This command makes sure *named* is in the proper directory when you try to include files by relative path names with $INCLUDE. It also allows *named* to run in a location that is reasonable to dump core if necessary.

Primary Master

The line in the boot file that designates a primary server for a zone looks like this:

```
primary      Berkeley.EDU    named.hosts
```

The first field specifies that the server is a primary one for the zone stated in the second field. The third field is the name of the file from which the data are read.

Secondary Master

The line for a secondary server is similar to the primary except that it lists addresses of other servers (usually primary servers) from which the zone data will be obtained. For example:

```
secondary  Berkeley.EDU 128.32.0.10 128.32.0.4 ucbhosts.bak
```

The first field specifies that the server is a secondary master server for the zone stated in the second field. The two network addresses specify the name servers that are primary for the zone. The secondary server gets its data across the network from the listed servers. It tries each server in the order listed until it successfully receives the data from a listed server.

If a file name is present after the list of primary servers, data for the zone will be saved in that file. When the server first starts, it loads the data from the backup file if possible, and then consults a primary server to check that the zone information is still up-to-date.

Caching-Only Server

All servers should have a line as follows in the boot file to prime the name server's cache:

```
cache              .          root.cache
```

All cache files listed will be read in at *named* start-up time. Any values still valid will be reinstated in the cache, and the root name server information in the cache files will always be used to handle initial queries.

The name server needs to know the servers that are the authoritative name servers for the root domain of the network. The *root.cache* file primes the server's cache with the addresses of these higher authorities. This file uses the Standard Resource Record format (or Master File format) described later in this chapter.

You do not need a special line to designate that a server is a caching server. What denotes a caching-only server is the absence of authority lines, such as *secondary* or *primary* in the boot file.

Forwarders

Any server can make use of *forwarders*, i.e., another server capable of processing recursive queries to try resolving queries on behalf of other systems. The *forwarders* command specifies forwarders by Internet address as:

```
forwarders        128.32.0.10        128.32.0.4
```

There are two main reasons to use forwarders: First, your system may not have full network access and cannot send IP packets to the rest of the network. Therefore, it must rely on a forwarder with access to the full net. Second, the forwarder sees a union of all queries as they pass through the server and therefore builds up a fuller cache of data than the cache in a typical workstation name server. In effect, the *forwarder* becomes a meta-cache that all hosts can benefit from, thereby reducing the total number of queries from that site to the rest of the net.

Slave Mode

You can use slave mode if forwarders are the only possible way to resolve queries due to lack of full net access. You can also use slave mode if you wish to prevent the name server from using other than the listed forwarders. Slave mode is activated by the following command in the boot file:

```
slave
```

If you use *slave*, you must specify forwarders. In slave mode, the server forwards each query to each of the the forwarders until an answer is found or the list of forwarders is exhausted.

9.4.3 Domain Data Files

A typical *named* setup has three standard files to specify the data for a domain. In the examples, below they are called *named.hosts*, *named.rev*, and *localhost.rev*. If you have more than one domain, incorporate the domain name as part of the file name when you create your versions of these files. The files in the Examples directory must be changed to reflect your setup. These files use the Standard Resource Record Format described in the next subsections. The location of the files is specified in the boot file.

named.hosts

This file contains all the data about the machines in this zone.

named.rev

This file specifies the IN-ADDR.ARPA domain, which is used to translated IP addresses into host names. As Internet host addresses do not fall within domain boundaries, this special domain was formed to allow inverse mapping. The IN-ADDR.ARPA domain for a host has four labels preceding it. These labels correspond to the 4 octets of an Internet address. All four octets must be specified, even if an octet is zero. For example, the Internet address 128.32.130.12 is located in the domain 12.130.32.128.IN-ADDR.ARPA. This reversal of the address allows the natural grouping of hosts in a network. An IN-ADDR.ARPA domain can also represent a network. For example, the ARPANET was net 10. That means there was a domain called 10.IN-ADDR.ARPA.

localhost.rev

This file specifies the IN-ADDR.ARPA domain of the local loopback interface's network address, 127.0.0.1. The address is better known as "localhost" address. Many important network programs depend on the information in this domain.

9.5 Standard Resource Record Format

The records in the name server data files are called resource records. The Standard Resource Record (RR) Format is specified in RFC-1035. The standard format of resource records is:

```
{name}  {ttl}  addr-class  Record Type  Record Specific data
```

The first field is the name of the domain record. It must always start in column 1. For some RRs the name may be left blank, in which case it takes on the name of the previous RR. The second field is an optional time-to-live field, which specifies how long this data will be stored in the database. When this field is blank, the default time-to-live is specified in the Start of Authority (*SOA*) resource record (described latter in this chapter). The third field is the address class. Currently only the *IN* class (for Internet hosts and addresses) is recognized. The fourth field states the type of the resource record. The fields after that depend on the type of RR.

Case is preserved in names and data fields when loaded into the name server. All comparisons and look-ups in the name server database are case insensitive.

If you specify TTLs for resource records, it is important that they are set to appropriate values. The TTL is the time (in seconds) that a resolver will use the data it got from your server before it asks your server again. If you set the value too low, your server will get loaded down with lots of repeat requests. If you set it too high, then information you change will not get distributed in a reasonable amount of time.

Most host information does not change much over long time periods. A good way to set up your TTLs would be to set them at a high value, and then lower the value if you know a change will be coming soon. You might set most TTLs to anywhere between a day (86400) and a week (604800). Then, if you know some data will be changing in the near future, set the TTL for that RR down to a lower value (an hour to a day) until the change takes place, and then put it back up to its previous value. Also, all resource records with the same name, class, and type should have the same TTL value.

The following characters have special meanings:

`.` A free-standing dot in the name field refers to the current domain.

`@` A free-standing @ in the name field denotes the current origin.

`..` Two free-standing dots represent the null domain name of the root when used in the name field.

`\`*x* The backslash designates that the special meaning of the character *x* does not apply. The *x* represents any character other than a digit (0-9). For example, use `\.` to place a dot character in a label.

`\`*DDD* Where each *D* is a digit, is the octet corresponding to the decimal number described by *DDD*. The resulting octet is assumed to be text and is not checked for special meaning.

`( )` Parentheses are used to group data that cross a line. In effect, newlines are not recognized within parentheses. Useful with SOA and WKS records.

`;` Semicolon starts a comment; the remainder of the line is ignored.

`*` An asterisk is a wildcard character.

Most resource records will have the current origin appended to names if they are not terminated by a "." (period character). This is useful for appending the current domain name to the data, such as machine names, but can cause problems if you do not want this to happen. If the name is not in the domain for which you are creating the data file, end the name with a ".". However, do not append the period to Internet addresses.

$INCLUDE

An include line begins with $INCLUDE, starting in column 1, and is followed by a file name. This feature helps you use multiple files for different types of data. For example:

```
$INCLUDE /usr/etc/named.d/mailboxes
```

This line is a request to load the file */usr/etc/named.d/mailboxes*. The $INCLUDE command does not cause data to be loaded into a different zone or tree. It allows data for a given zone to be organized in separate files. For example, mailbox data might be kept separately from host data using this mechanism.

$ORIGIN

The origin is a way of changing the origin in a data file. The line starts in column 1, and is followed by a domain origin. This is useful for putting more then one domain in a data file.

```
$ORIGIN  Berkeley.EDU.
```

SOA – Start Of Authority

name	*{ttl}*	*addr-class*	*SOA*	*Source*	*Person-in-charge*
@		IN	SOA	ucbvax.Berkeley.EDU.	kjd.ucbvax.Berkeley.EDU. (
			1.1	; Serial	
			10800	; Refresh	
			3600	; Retry	
			3600000	; Expire	
			86400	; Minimum	
			)		

The Start of Authority, *SOA*, record designates the start of a zone. There should only be one *SOA* record per zone. The *name* is the name of the zone. It can be a complete domain name like "Berkeley.EDU." or a name relative to the the current $ORIGIN. The "@" indicates the current zone name, taken from the "primary" line in the *named.boot* file or from a previous $ORIGIN line. *Source* is the name of the host on which the master data file resides, typically the primary master server. *Person-in-charge* is the mailing address for the person responsible for the name server. The serial number is the version number of this data file. This number should be incremented whenever a change is made to the data. The name server cannot handle numbers over 9999 after the decimal point. A useful convention is to encode the current date in the serial number. For example, 25 April 1990 edit #1 is encoded as:

```
90042501
```

Increment the edit number if you modify the file more than once that day. The refresh indicates how often, in seconds, a secondary name server is to check with the primary name server to see if an update is needed. The retry indicates how long, in seconds, a secondary server is to retry after a failure to check for a refresh. Expire is the maximum number of seconds that a secondary name server has to use the data before they expire for lack of getting a refresh. Minimum is the default number of seconds to be used for the time-to-live field on resource records with no explicit time-to-live value.

NS – Name Server

{name}	*{ttl}*	*addr-class*	*NS*	*Name servers name*
		IN	NS	ucbarpa.Berkeley.EDU.

The Name Server record, *NS*, lists the name of a machine that provides domain service for a particular domain. The name associated with the RR is the domain name and the data portion is the name of a host that provides the service. Note that the machines providing name service do not have to live in the named domain. There should be one *NS* record for each master server (primary or secondary) for the domain. Note that more than approximately 10-15 NS records for a zone might exceed DNS datagram size limits.

NS records for a domain must exist in both the zone that delegates the domain, and in the domain itself. If the name server host for a particular domain is itself inside the domain, then a "glue" record will be needed. A glue record is an A (address) RR that specifies the address of the server. Glue records are only needed in the server delegating the domain, not in the domain itself. If for example the name server for domain SRI.COM was KL.SRI.COM, then the NS and glue A records on the delegating server would look like this:

```
SRI.COM.      IN   NS      KL.SRI.COM.
KL.SRI.COM.   IN   A       10.1.0.2
```

The administrators of the delegating and delegated domains should insure that the NS and glue RRs are consistent and remain so.

A – Address

{name}	*{ttl}*	*addr-class*	*A*	*address*
ucbvax		IN	A	128.32.133.1
		IN	A	128.32.130.12

The Address record, *A*, lists the address for a given machine. The name field is the machine name, and the address is the network address. There should be one *A* record for each address of the machine.

HINFO – Host Information

{name}	*{ttl}*	*addr-class*	*HINFO*	*Hardware*	*OS*
		IN	HINFO	SGI-IRIS-4D/380VGX	IRIX

The Host Information resource record, *HINFO*, is for host-specific data. This record lists the hardware and operating system running at the listed host. Only a single space separates the hardware information and the operating system information. To include a space in the machine name, you must place quotes around the name. There should be one *HINFO* record for each host. See the file */usr/etc/named.d/README* for the current list of names for IRIS-4D Series workstations and servers. The "Assigned Numbers" RFC contains names for other types of hardware and operating systems.

WKS – Well-Known Services

{name}	*{ttl}*	*addr-class*	*WKS*	*address*	*protocol*	*list of services*
		IN	WKS	192.12.63.16	UDP	who route timed domain
		IN	WKS	192.12.63.16	TCP	(echo telnet chargen ftp smtp time domain bootp finger sunrpc)

The Well-Known Services record, *WKS*, describes well-known services supported by a particular protocol at a specified address. The list of services and port numbers comes from the list of services specified in */etc/services*. There should be only one *WKS* record per protocol per address.

CNAME – Canonical Name

aliases	*{ttl}*	*addr-class*	*CNAME*	*Canonical name*
ucbmonet		IN	CNAME	monet

The Canonical Name resource record, *CNAME*, specifies an alias or nickname for the official, or canonical, host name. This record should be

the only one associated with the alias name. All other resource records should be associated with the canonical name, not with the nickname. Any resource records that include a domain name as their value (e.g., NS or MX) should list the canonical name, not the nickname.

Nicknames are also useful when a host changes its name. In that case, it is usually a good idea to have a CNAME record so that people still using the old name will get to the right place.

PTR – Domain Name Pointer

name	*{ttl}*	*addr-class*	*PTR*	*real name*
6.130		IN	PTR	monet.Berkeley.EDU.

A Domain Name Pointer record, *PTR*, allows special names to point to some other location in the domain. The above example of a *PTR* record is used to set up reverse pointers for the special *IN-ADDR.ARPA* domain. This line is from the example *named.rev* file. *PTR* names should be unique to the zone. Note the trailing period (.) on the real name to prevent *named* from appending the current domain name.

MB – Mailbox

name	*{ttl}*	*addr-class*	*MB*	*Machine*
ben		IN	MB	franklin.Berkeley.EDU.

The Mailbox record, *MB*, lists the machine where a user receives mail. The name field is the user's login. The machine field lists the machine to which mail is to be delivered. Mail box names should be unique to the zone.

MR – Mail Rename Name

name	*{ttl}*	*addr-class*	*MR*	*corresponding MB*
Postmaster		IN	MR	ben

The Mail Rename record, *MR*, lists aliases for a user. The name field lists the alias for the name listed in the fourth field, which should have a corresponding *MB* record.

MINFO – Mailbox Information

name	*{ttl}*	*addr-class*	*MINFO*	*requests*	*maintainer*
BIND		IN	MINFO	BIND-REQUEST	kjd.Berkeley.EDU.

The Mail Information record, *MINFO*, creates a mail group for a mailing list. This resource record is usually associated with a Mail Group (*MG*), but can be used with a Mail Box (*MB*) record. The *name* is the name of the mailbox. The *requests* field is where mail such as requests to be added to a mail group should be sent. The *maintainer* is a mailbox that should receive error messages. This is appropriate for mailing lists when errors in members' names should be reported to someone other than the sender.

MG – Mail Group Member

{mail group name}	*{ttl}*	*addr-class*	*MG*	*member name*
		IN	MG	Bloom

The Mail Group record, *MG*, lists members of a mail group. An example for setting up a mailing list follows:

```
Bind      IN   MINFO   Bind-Request          kjd.Berkeley.EDU.
          IN   MG      Ralph.Berkeley.EDU.
          IN   MG      Zhou.Berkeley.EDU.
```

MX – Mail Exchanger

name	*{ttl}*	*addr-class*	*MX*	*preference value*	*mailer exchanger*
Munnari.OZ.AU.		IN	MX	10	Seismo.CSS.GOV.
*.IL.		IN	MX	10	CUNYVM.CUNY.EDU.

The Mail Exchanger record, *MX*, specifies a machine that can deliver mail to a machine that is not directly connected to the network. In the first example above, Seismo.CSS.GOV. is a mail gateway that can deliver mail to Munnari.OZ.AU. Other machines on the network cannot deliver mail

directly to Munnari. The two machines, Seismo and Munnari, can have a private connection or use a different transport medium. The preference value is the order that a mailer should follow when there is more then one way to deliver mail to a single machine. See RFC-974 for more detailed information.

Wildcard names containing the character "*" can be used for mail routing with *MX* records. Servers on the network can state that any mail to a domain is to be routed through a relay. In the second example above, all mail to hosts in the domain IL is routed through CUNYVM.CUNY.EDU. This is done by creating a wildcard resource record, which states that *.IL has an *MX* of CUNYVM.CUNY.EDU.

Also see Partridge, C., "Mail Routing and The Domain System." *Internet Request For Comment 974*, Network Information Center, SRI International, Menlo Park, California. February 1986.

9.6 Management

This section contains information about the steps to maintain the databases, and enabling, disabling and controlling *named.*

To add a new host to your zone files:

1. Edit the appropriate zone file for the domain the host is in.
2. Add an entry for each address of the host.
3. Optionally add CNAME, HINFO, WKS, and MX records.
4. Add the reverse IN-ADDR entry for each host address in the appropriate zone files for each network the host in on.

To delete a host from the zone files:

1. Remove all the host's resource records from the zone file of the domain the host is in.
2. Remove all the host's PTR records from the IN-ADDR zone files for each network the host was on.

To add a new subdomain to your domain:

1. Setup the other domain server and/or the new zone file.
2. Add an NS record for each server of the new domain to the zone file of the parent domain.
3. Add any necessary glue RRs, as described in the NS section.

9.6.1 /etc/config/named

Named is started automatically during system startup if the configuration flag */etc/config/named* is "on." Use the command:

```
/etc/chkconfig named on
```

to enable the name server, and

```
/etc/chkconfig named off
```

to disable it at the next system startup. To terminate the program immediately, use:

```
/etc/killall -KILL named
```

9.6.2 /etc/config/named.options

This file is optional. It is used during system startup and by *named.restart*. Specify command line arguments for *named* in this file. See *named*(1M) for details on the options.

9.6.3 /usr/etc/named.reload

This shell script sends the HUP signal to *named*, which causes it to read *named.boot* and reload the database. All previously cached data are lost. This is useful when you have made a change to a data file and you want *named*'s internal database to reflect the change.

9.6.4 /usr/etc/named.restart

This shell script terminates the running *named* and starts a new one.

9.7 Debugging

When *named* is running incorrectly, check first in */usr/adm/SYSLOG* for any messages. For additional information, send it one of the following signals using *killall*(1M):

```
/etc/killall -SIG named
```

where *SIG* is INT, ABRT, USR1 or USR2.

INT — Dumps the current database and cache to */usr/tmp/named_dump.db*. This should indicate whether the database was loaded correctly.

ABRT — Dumps statistics data into */usr/tmp/named.stats*. Statistics data are appended to the file.

USR1 — Turns on debugging. Each following USR1 increments the debug level. There are 10 debug levels, and each prints more detailed information. A debug level of 5 is useful for debugging lookup requests. The output goes to */usr/tmp/named.run*.

USR2 — Turns off debugging completely.

9.7.1 nslookup

The *nslookup*(1) command is a useful debugging tool to query local and remote name servers. *nslookup* has two modes: interactive and non-interactive. Interactive mode allows the user to query the name server for information about various hosts and domains or print a list of hosts in the domain. Non-interactive mode is used to print just the name and requested information for a host or domain. The following example gets the address record for the host monet.berkeley.edu.

```
% nslookup
Default Server:  ucbvax.berkeley.edu
Address:  128.32.133.1

> monet
Server:  ucbvax.berkeley.edu
Address:  128.32.133.1

Name:    monet.berkeley.edu
Address:  128.32.130.6
```

To exit, type <Ctrl-D> or "exit." The "help" command summarizes the available commands. The complete set of commands are described in the *nslookup*(1) manual page.

9.8 Sample Files

The following section contains sample files for the name server, including sample boot files for the different types of servers and sample domain database files.

9.8.1 Primary Master Server Boot File

```
; type          domain                  source file or host
;
directory       /usr/etc/named.d
primary         Berkeley.EDU            named.hosts
primary         32.128.in-addr.arpa     named.rev
primary         0.0.127.in-addr.arpa    localhost.rev
cache           .                       root.cache
```

9.8.2 Secondary Master Server Boot File

```
; type          domain                  source file or host
;
directory       /usr/etc/named.d
secondary       Berkeley.EDU            128.32.130.11 128.32.130.12 ucbhosts.bak
secondary       32.128.in-addr.arpa     128.32.130.11 128.32.130.12 ucbhosts.rev.bak
primary         0.0.127.in-addr.arpa    localhost.rev
cache           .                       root.cache
```

9.8.3 Caching-Only Server Boot File

```
; type          domain                  source file or host
;
directory       /usr/etc/named.d
cache           .                       root.cache
primary         0.0.127.in-addr.arpa    localhost.rev
```

9.8.4 Client resolv.conf

```
domain Berkeley.EDU
nameserver 128.32.130.11
nameserver 128.32.130.12
```

9.8.5 root.cache

```
; Initial cache data for root domain servers.

.                       IN  NS  NS.NICNS.DDN.MIL.
                        IN  NS  A.ISI.EDU.
                        IN  NS  AOS.BRL.MIL.
                        IN  NS  C.NYSER.NET.
                        IN  NS  GUNTER-ADAM.AF.MIL.
                        IN  NS  NS.NASA.GOV.
                        IN  NS  TERP.UMD.EDU.
; Prep the cache.  Order does not matter
NS.NIC.DDN.MIL.         IN  A   192.67.67.53
A.ISI.EDU.              IN  A   26.3.0.103
                        IN  A   128.9.0.107
AOS.BRL.MIL.            IN  A   128.20.1.2
                        IN  A   192.5.25.82
C.NYSER.NET.            IN  A   192.33.4.12
GUNTER-ADAM.AF.MIL.     IN  A   26.1.0.13
NS.NASA.GOV.            IN  A   128.102.16.10
                        IN  A   192.52.195.10
TERP.UMD.EDU.           IN  A   128.8.10.90
```

9.8.6 localhost.rev

```
@   IN  SOA     ucbvax.Berkeley.EDU. kjd.ucbvax.Berkeley.EDU. (
                1.2 ; Serial
                3600 ; Refresh
                300 ; Retry
                3600000 ; Expire
                14400 ; Minimum
                )
    IN  NS      ucbvax.Berkeley.EDU.
0   IN  PTR     loopback.ucbvax.Berkeley.EDU.
1   IN  PTR     localhost.
```

9.8.7 named.hosts

```
@           IN   SOA     ucbvax.Berkeley.EDU. kjd.monet.Berkeley.EDU. (
                         1.1 ; Serial
                         10800 ; Refresh
                         3600 ; Retry
                         3600000 ; Expire
                         86400  ; Minimum
                         )
            IN   NS      ucbarpa.Berkeley.EDU.
            IN   NS      ucbvax.Berkeley.EDU.
localhost   IN   A       127.1
ucbarpa     IN   A       128.32.130.11
            IN   HINFO   VAX-11/780 UNIX
arpa        IN   CNAME   ucbarpa
monet       IN   A       128.32.130.6
ucbvax      IN   A       128.32.133.1
            IN   A       128.32.130.12
            IN   HINFO   VAX-11/750 UNIX
            IN   WKS     128.32.130.12 UDP (
                         syslog route timed domain )
            IN   WKS     128.32.130.12 TCP (
                         telnet sunrpc ftp finger
                         smtp domain nameserver )
ucb-vax     IN   CNAME   ucbvax
toybox      IN   A       128.32.131.119
            IN   HINFO   Pro350 RT11
toybox      IN   MX      10  monet.Berkeley.EDU
miriam      IN   MB      vineyd.DEC.COM.
postmistress IN  MR      Miriam
Bind        IN   MINFO   Bind-Request kjd.Berkeley.EDU.
            IN   MG      Ralph.Berkeley.EDU.
            IN   MG      Zhou.Berkeley.EDU.
```

9.8.8 named.rev

```
@       IN      SOA     ucbvax.Berkeley.EDU. kjd.monet.Berkeley.EDU. (
                        1.1 ; Serial
                        10800 ; Refresh
                        3600 ; Retry
                        3600000 ; Expire
                        86400 ; Minimum
                        )
        IN      NS      ucbarpa.Berkeley.EDU.
        IN      NS      ucbvax.Berkeley.EDU.
0.0     IN      PTR     Berkeley-net.Berkeley.EDU.
        IN      A       255.255.255.0
0.130   IN      PTR     csdiv-net.Berkeley.EDU.
11.130  IN      PTR     ucbarpa.Berkeley.EDU.
12.130  IN      PTR     ucbvax.Berkeley.EDU.
6.130   IN      PTR     monet.Berkeley.EDU.
```

Appendix A: The Mail System

The mail system is a group of programs that you can use to send messages and receive messages from other users on the network. You can send mail through either UUCP or TCP/IP. The IRIX operating system uses System V */bin/mail*, 4.3BSD */usr/sbin/Mail*, and *sendmail* for its mail implementation.

This appendix is for a system administrator for who sets up and maintains the mail system on the host or network.

A.1 Mail System Hierarchy

The mail system programs can be divided into four functional categories:

User Interfaces
: These programs provide the user interface for the creation of new messages and the reading, removal, and/or archival of received messages. Examples of this level of the mail system are the *mail_att*(1) and *mail_bsd*(1) programs.

Mail Routing
: These programs route the messages through the network to the appropriate systems. An example is the *sendmail*(1M) program which routes messages between various user interface, transmission and delivery programs.

Mail delivery
: These programs deposit mail into a data file for later perusal by a user or another program. An example of this type of program is */bin/mail –d* which is used by *sendmail*(1M) to deliver local mail.

Mail Transmission

These programs are responsible for transmitting messages from one host to another. Mail transmission is used when the mail destination resides on a remote host. Examples of this level of the mail system are UUCP, which uses its own protocols and runs over serial lines, and the section of the *sendmail* program, which implements the Simple Mail Transmission Protocol (SMTP) over TCP/IP. (Note that for TCP/IP mail, the *sendmail* program acts as an integrated routing and transmission program.) In all cases, the mail transmission process has a counterpart — the mail reception process. In most cases, both processes reside in the same program.

After you compose a message using an available user interface program, the message is sent to a mail routing program. The routing program is responsible for determining the destination of the message and calling the appropriate delivery program (for mail to a user on the local host) or transmission program (for mail to a user on a remote host). Likewise, when a mail reception process part of a transmission program receives a message from another host, it sends it to the local mail routing program which will eventually call the appropriate delivery or transmission program. (Note that the latter would be called in the event that the local host was acting as a relay between two other hosts which wished to communicate.)

When you send a mail message on a network that uses TCP/IP, several layers of network software are involved. The layers of TCP/IP mail network software look like this:

SMTP/sendmail
TCP
IP
Network

The Transmission Control Protocol (TCP) layer supports the SMTP protocol which the *sendmail* program uses to transmit mail to other TCP/IP hosts. The *sendmail* program is responsible for calling local delivery programs, mail routing and TCP/IP mail transmission. Note that *sendmail* uses a separate UUCP transmission program to handle messages to UUCP hosts.

A.2 Examples Using Mail

To mail a message to a user on another system, you first must determine the user's network address. The next two sections illustrate sample mail messages. The first shows a message sent without routing. The second message shows a message routed through UUCP. Routing is a means of sending a message to the desired host. You can configure your *sendmail.cf* file to automatically route your mail (more on this later).

A.2.1 Mail without Routing

The following is an example of a common host interconnection. Hosts *aspen*, *elm*, and *willow* are on an Ethernet connection and use SMTP (the Simple Mail Transmission Protocol). Host *oak* is connected to host *willow* by a serial line, and can communicate only through UUCP. The following figure shows four users: *mike* on host *aspen*, *alice* on host *elm*, *joe* on host *willow*, and *barbara* on host *oak*.

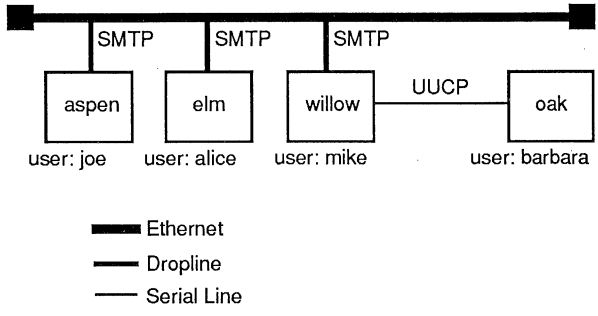


Figure A-1. Map of Users and Hosts

If user *mike* wants to send mail to *alice* at host *elm*, *mike* types:

```
% Mail alice@elm
Subject: Quarterly report

Please send me the figures for production last month.
Thanks,

Mike


<ctrl-d>
%
```

When *Mail* is finished composing the message, it calls *sendmail.* The *sendmail* program, via its configuration file (*sendmail.cf*) figures out how to send the mail to *alice* at host *elm*, and which of its resources can perform the transmission. In this simple case, because hosts *aspen* and *elm* are on the same network, and because they both use SMTP, *sendmail* tries to deliver the mail directly.

The *sendmail* on host *aspen* assumes that a daemon which understands SMTP exists on host *elm* and attempts to open a connection with it. If the connection is successful, the mail is transmitted to host *elm.*

The *sendmail* program on host *elm* receives the SMTP message and runs through its algorithm (as described in *sendmail.cf*) to resolve the destination address. For this example, sendmail *on host elm* will recognize that the host part of the address is the local host, and that *alice* is a user on this host and resolves the address to just *alice.* The local mail delivery program, */bin/mail –d*, is then called to deposit the mail into *alice*'s mailbox.

A.2.2 Mail with Routing

Suppose *alice* on host *elm* wants to send mail to *barbara* on host *oak:*

```
%  Mail  barbara@oak
Subject: Can I use your printer?

Is it OK if I use your printer all day tomorrow?

Thanks,
alice
<ctrl-d>
%
```

Mail composes the message and hands it to *sendmail* to route. The *sendmail* program on host *elm*, via its configuration file (*sendmail.cf*) determines that it cannot send the mail directly to host *oak*. Instead, the mail must be routed through host *willow*, because host *willow* has the only direct connection to host *oak*. The *sendmail* program rewrites the mail address to *@willow:barbara@oak,* (read as "send to host *willow* for *barbara* at host *oak*") then sends the message to the SMTP server on host *willow*. Host *willow* strips off its own address, resulting in the address *barbara@oak*.

Note at this point that not all mailers use the same address format. UUCP does not use the same format as TCP/IP. To cope with this, mailer-specific configurations within *sendmail.cf* direct *sendmail* to rewrite addresses into the correct form for each mailer.

Once the *barbara@oak* address is received at host *willow*, the *sendmail.cf* file for host *willow* directs *sendmail* to use UUCP to send mail to host *oak*. Note that this process is transparent to *alice*, the originator of the message. Once *sendmail* decides that UUCP must be used to transmit the mail, it rewrites the address to *oak!barbara* and calls the UUCP mail transmission program as instructed by the data in the *sendmail.cf* file. UUCP then takes the message and mails it to *barbara* via *rmail* (*rmail* is responsible only for delivering UUCP mail). On host *oak*, UUCP uses *rmail* to receive the message, and *rmail* uses *sendmail* to deliver it. Once *sendmail* has the message, it determines that no more forwarding is needed and uses the local mailer to deliver the mail; *barbara* receives the message.

Note that *alice* could also have routed the mail explicitly by typing:

```
Mail  willow\!oak\!barbara
```

A.3 The *sendmail.cf* File

The */usr/lib/sendmail.cf* file contains rules for parsing destination addresses and for rewriting addresses for different mail delivery programs. It also contains information about network configuration, and rules for routing messages to other hosts. The *sendmail.cf* file is a control file for the *sendmail* program. IRIX provides a *sendmail.cf* file suitable for most uses. For more information on *sendmail* and the *sendmail.cf* file, see the *Mail Reference Guide*. For specifics about configuring your *sendmail.cf* file, see the comments at the top of the */usr/lib/sendmail.cf* file itself.

A.4 Address Aliasing

You use address aliasing to equate one address with another. It is a form of shorthand: you can enter a short address that takes the place of an entire routing address.

Some mail user interface programs provide an aliasing feature for destination addresses. Unfortunately, not all do. To provide a common aliasing facility, *sendmail* supports aliasing directly. *sendmail*'s aliasing is invoked after the user interface programs' aliasing, so some aliases may have conflicting meanings. Make sure the meanings are resolved before using the aliasing. The file */usr/lib/aliases* contains a text form of a database used for mail aliasing; (if you use the Yellow Pages (YP), then you use the "aliases" YP database).

This example shows an entry in */usr/lib/aliases*:

```
alice: elm!alice
```

When you type *alice* as an address, *sendmail* routes the message to *elm!alice*. The message travels through system *elm* to user *alice*.

The next example shows a mail message that uses the alias.

```
%  Mail alice
Can we meet at nine tomorrow?

Barbara
<ctrl-d>
%
```

Aliasing takes place only on the host machine from which you originally send the mail message. If you want to redirect incoming mail, use a *.forward* file. When a message is received, *sendmail* looks for a *.forward* file in the recipient's home directory. If *sendmail* finds that file, it forwards the message to the address that the file contains. If mail is forwarded, *sendmail* applies the same rules that it applied to the original recipient address, and restarts the routing process.

A.5 UUCP and *sendmail*

The *sendmail* program uses UUCP to deliver mail to any system that can be reached through UUCP.

UUCP has its own routing system. The standard *sendmail.cf* configuration file uses the contents of */usr/lib/uucp/Systems* to create a class of recognized hosts to which mail may be sent directly via UUCP. The *sendmail* program also supports access the local UUCP path alias database via the *$P* macro and the *$[!* pathalias lookup operator in the *sendmail.cf* configuration file. Please see the rules pertaining to UUCP routing in the */usr/lib/sendmail.cf* file for more details.

In order for UUCP to function in the mail program, all systems that have only UUCP on the local network must be listed in the UNIX system file */usr/lib/uucp/Systems*. You must become the superuser to edit this file.

Search the */usr/lib/uucp/Systems* file for this line:

```
dagwood  Any  dagwood  Any  unused  ogin:-BREAK-ogin:
   uucp  ssword:  secret
```

Modify this line to make it the same as the corresponding line in your */etc/passwd* file. Change *uucp* in the line above to your user name, and change *ssword* to your UUCP password, if you have one.

A.6 Relevant Documentation

You can find useful information to help you plan and set up your electronic mail system in these documents:

- "Communication Tutorial" in the *IRIS-4D User's Guide*
- "Basic Networking" in the *IRIX System Administrator's Guide.*
- *Mail Reference Manual*
- *mail_att*(1) for *mail* description, and *mail_bsd*(1) for *Mail* description in the *IRIX User's Reference Manual.*

Index

C

D

E

F

G

H

I

O

P

R

S

T

U

V

W

X

Y