# IRIS Communications Guide

*Version 1.1*

**Silicon Graphics, Inc.**
**2011 Stierlin Road**
**Mountain View, CA 94043**

**Document Number 007-0390-010**
(includes *IRIS Communications Guide Update Package*
Document Number 007-0390-011)

**Technical Publications:**

Gail Kesner
Diane M. Wilford

**Engineering:**

Vernon Schryver
Brendan Eich
Paul Mielke
Kipp Hickman



**IRIS Communications Guide**
**Document Number 007-0390-010**
(includes *IRIS Communications Guide Update Package*
Document Number 007-0390-011)

CONTENTS

# 1. Introduction

IRIS terminals and workstations are powerful tools. The ability to network them with one another and with other systems greatly enhances their power. Silicon Graphics, Inc., provides the IRIS with networking flexibility through standard or optional software programs and hardware.

The *IRIS Communications Guide* is an overview of the communications capabilities of the IRIS. IRISes can be networked with one another and with a variety of hosts; they can function in either a TCP/IP or XNS environment; and they can accommodate a number of applications, including file transfer and electronic mail. Table 1-1 shows the networking possibilities with the IRIS.

| IRIS | Host | Method of Communication |
|---|---|---|
| Terminal | IRIS workstation | TCP/IP or XNS protocols, or serial line |
| | DEC VAX running UNIX or VMS | TCP/IP or XNS protocols, or serial line |
| | IBM running VM/CMS or OS/MVS | Geometry Link |
| | UNIX host | UUCP |
| | UNIX host running NFS | TCP/IP protocols |
| Workstation | IRIS workstation | TCP/IP or XNS protocols, or serial line |
| | DEC VAX running UNIX or VMS | TCP/IP or XNS protocols, or serial line |
| | IBM running VM/CMS or OS/MVS | Geometry Link |
| | UNIX host | UUCP |
| | UNIX host running NFS | TCP/IP protocols |
| | UNIX or VMS host running mail program | Mail |

Table 1-1: Communication with the IRIS

This guide addresses five kinds of communication:

- Interactive communication, in which commands entered locally are executed on a remote system.
- Communication between the IRIS as a graphics terminal and a remote host. In graphics terminal mode, IRIS terminals and workstations are used as graphics output devices. The graphics programs usually run and are maintained on the host.
- File transfer among devices on the same network or on different networks.
- Dynamic file access across a network using Network File System (NFS). NFS allows users to mount directories from a remote host and treat remote files as if they were local.
- Electronic mail between hosts.

The *IRIS Communications Guide* is divided into the following sections:

Chapter 1: *Introduction*

Chapter 2: *Communications Protocols*
Describes the IRIS as it functions in TCP/IP and XNS environments.

Chapter 3: *Host Communications*
Describes the hardware and software that enables the IRIS to communicate with a variety of hosts.

Chapter 4: *Network File System*
Introduces NFS, the system that allows files to be shared across the network.

Chapter 5: *The UUCP Program*
Describes the utility that enables communication between the IRIS and remote systems.

Chapter 6: *The Mail System*
Describes the mail system, which allows users on the network to send messages to one another.

Appendix A: *Communications Options on the IRIS*
Describes the communications options on the IRIS series 2000 and 3000.

Appendix B: *Bibliography*
Lists further reading material on topics covered in this manual.

Appendix C: *NFS Reference Material*
Introduces and presents specifications for External Data Representation Protocols and Remote Procedure Call Protocols.

Appendix C: *NFS Reference Material*
Introduces and presents specifications for External Data Representation Protocols and Remote Procedure Call Protocols.

Appendix D: *Mail Systems and Addressing in 4.2bsd*
Describes mail routing and *sendmail* in 4.2bsd

Appendix E: *SENDMAIL — An Internetwork Mail Router*
Describes *sendmail* operations in internetwork mail routing

Appendix F: *SENDMAIL — Installation and Operation Guide*
Describes installation, normal operations, and tailoring for the mail configuration files.

Appendix G: *The Domain Naming Convention for Internet User Applications*
Describes the hierarchical naming convention based on domain concepts.

Because the *IRIS Communications Guide* is an overview, most of the topics covered are described more fully in other manuals. Refer to Appendix B for a complete list of references.

This document uses the standard UNIX convention for referring to entries in the UNIX documentation. The entry name is followed by a section number in parentheses. For example, *cc*(1) refers to the *cc* manual entry in Section 1 of the *UNIX Programmer's Manual, Volume I.*

In command syntax descriptions, square brackets surrounding an argument indicate that the argument is optional. Words in *italics* represent variable parameters, which should be replaced with the string or value appropriate for the application. Examples are shown in `typewriter font`.

In text descriptions, filenames and UNIX commands are written in *italics.*

# 2. Communications Protocols

A communications protocol is a procedure with a well-defined format that allows two or more systems to communicate over a physical link. The International Standards Organization recognizes a few standard protocols. The IRIS can run with two of these: Transmission Control Protocol/Internet Protocol (TCP/IP) and Xerox Network Systems (XNS). Although systems running different protocols can coexist on the same network, communication is possible only among systems running the same protocols.

The TCP/IP or XNS protocols can be used for all IRIS connections except those in IBM environments (refer to Section 3.3 for a discussion of the IBM connections). On the IRIS, the TCP/IP and XNS protocol sets reside in separate kernels. To change from one to the other, you must become the superuser, issue the *kernel*(1M) command with the desired protocol argument (**tcp** or **xns**), and reboot the system. For more detailed information on the TCP/IP and XNS protocols, refer to the *TCP/IP User's Guide* and the *XNS User's Guide*.

This section provides an overview of the two protocol sets, including configuration of the IRIS for the protocol, and a brief description of the network commands that can be used with the protocols.

## 2.1 Using the IRIS with TCP/IP

Beginning with software releases GL2-W2.5 and GL2-W3.5, TCP/IP (based on 4.3bsd) is the standard protocol shipped with the IRIS workstations (2400, 2400 Turbo, 2500, 2500 Turbo, 3020, 3030). IRIS workstations running earlier software and all IRIS terminals use the XNS protocol set as the default. If another kernel is installed, you must use the *kernel*(1M) command with the **tcp** argument to install the TCP/IP kernel and reboot the system.

Each workstation on a network must have a unique host name. Refer to the *TCP/IP User's Guide* for instructions on configuring the host name and other system administration procedures required to set up an IRIS system in a TCP/IP environment.

The basic TCP/IP commands that can be issued at the IRIS are *rcp*, *rsh*, *rlogin*, *ftp*, and *telnet*. You must be running the TCP/IP kernel to use these commands. The daemon (a process running in the background) that supports these commands can be disabled on individual hosts for security (see the *inet*(1M) manual page). Following is a brief explanation of each command.

Refer to the *TCP/IP User's Guide* for a complete discussion of each one and for the other commands available with TCP/IP.

- The *rcp* command copies a file from one system running UNIX to another. The syntax of the command is:

  **rcp** [*sourcehost*:]*pathname* [*destinationhost*:]*pathname*

  If no host name is specified, UNIX assumes the local host.

- The *rsh* command connects your terminal to a remote host running UNIX, and allows you to execute the commands you issue. The syntax of the command is:

  **rsh** *hostname* [*commandname*]

  The command is executed on the specified host.

- The *rlogin* command initiates a login on a remote host running UNIX. The syntax of the command is:

  **rlogin** *hostname*

In an ARPANET environment, *telnet* and *ftp* perform the same functions as *rlogin* and *rcp*. The commands *rlogin* and *rcp* are faster and more reliable, and Silicon Graphics, Inc., recommends using these instead, if possible. Refer to the *TCP/IP User's Guide* for information on *telnet* and *ftp*.

## 2.2 Using the IRIS with XNS

IRIS workstations running software releases earlier than GL2-W2.5 and GL2-W3.5 and all IRIS terminals use the XNS protocol by default. Beginning with the GL2-W2.5 and GL2-W3.5 software release, the XNS protocol is an option on these systems.

IRIS terminals (2300, 2300 Turbo, 3010) using the XNS kernel require no host name configuration; they function only as terminals, not as hosts. IRIS workstations in an XNS environment must have a unique host name. Refer to the *XNS User's Guide* for the procedures required to set up an IRIS system in an XNS environment.

The UNIX operating system provides a set of commands that the IRIS can use in an XNS environment. You must be running the XNS kernel to use these commands. If another kernel is installed, you must use the *kernel*(1M) command with the **xns** argument to install the XNS kernel and reboot the system.

The basic commands in an XNS implementation that can be issued at the IRIS are *xcp*, *xx*, and *xlogin*. The daemon that supports these commands can be disabled on individual hosts for security (see the *xnsd*(1M) manual page). Following is a brief explanation of each command. Refer to the *XNS User's Guide* for a complete discussion of each one and for the other commands available with XNS.

- The *xcp* command copies a file from one system running UNIX to another. The syntax of the command is:

  **xcp** [*sourcehost:*]*pathname* [*destinationhost:*]*pathname*

  If no host name is specified for either the source or destination, UNIX assumes the local host. Typically, you have an account with the same user name on both hosts. If you have no account on the remote host, UNIX attempts to copy the specified file into or from the *guest* account (a default account) on the destination host.

- The *xx* command connects your terminal to a remote host running UNIX and allows you to execute the commands you issue. The syntax of the command is:

  **xx** *hostname* [*commandname*]

- The *xlogin* command initiates a login on a remote host running UNIX. The syntax of the command is:

  **xlogin** [*hostname*]

  When the connection is made, the remote host prompts for a login:

  ```
  login:
  ...
  ```

  If the normal disconnection commands (e.g., *logout*, CTRL-D ) do not function following an *xlogin* command, issue the following sequence:

  <CR> .

# 3. Host Communications

Silicon Graphics, Inc., provides standard and optional hardware and software that enable you to communicate with a variety of hosts.

## 3.1 The wsiris Program

The *wsiris*(1) program enables communication between an IRIS terminal and a host. It also allows an IRIS workstation to emulate an IRIS terminal. (For terminal emulation in IBM environments, you must use the *t3279*(1) program; refer to Section 3.3.) Before using *wsiris*, you must install the host software. The *IRIS Series 3000 Owner's Guide* lists the procedure for host software installation. For complete information on *wsiris*, see *wsiris*(1); the *IRIS Series 3000 Owner's Guide*; *IRIS Workstation Guide, Series 2000*; and *IRIS Terminal Guide, Series 2000*.

### 3.1.1 Terminal Operations

After loading UNIX from the default boot file and checking the file system, the IRIS 2300, 2300 Turbo, and 3010 terminals automatically load and run *wsiris*. The *wsiris* program enables the terminals to emulate standard ASCII character terminals. In addition, *wsiris* performs these functions:

- Sets up communication with the host
- Interprets graphics command sequences from the host

You can escape from *wsiris* to an interactive environment from which you can issue the UNIX commands necessary to perform basic system administration. However, the IRIS terminals do not have a complete UNIX operating system. (See the *IRIS Series 3000 Owner's Guide* and the *IRIS Terminal Guide, Series 2000*, for a description of the *wsiris* environment.)

After *wsiris* boots on the terminal, it displays the startup message:

```
IRIS GL2 Terminal Emulator
Connect to what host?
```

Enter the name of the host to which you want to connect. The login prompt from the host is displayed.

To perform system administration tasks, you need to exit *wsiris*. Issue the following command:

```
~!
```

### 3.1.2 Workstation Operations

The *wsiris* program allows a workstation to emulate an IRIS terminal while communicating with a remote host. The workstation can be configured to connect automatically to a specified host (see Section 5.2 in the *IRIS Terminal Guide, Series 2000* and Section 5.2 in the *Series 3000 Owner's Guide*). For IBM terminal emulation, refer to Section 3.3. To enter terminal emulation mode on a workstation, issue the *wsiris* command (see *wsiris*(1)). The workstation prompts for the host:

```
Connect to what host?
```

Enter the name of the host to which you want to connect. Your workstation is now in terminal emulation mode. You can run remote graphics programs from a remote host to your workstation or perform other functions, such as editing files, etc.

## 3.2 DEC VAX Running VMS

IRIS terminals and workstations can communicate with a DEC VAX running VMS in a TCP/IP or XNS environment. Before you can run the software that enables the IRIS to communicate with a VMS host, you must ensure that the appropriate protocol kernel is installed on the IRIS. To change kernels, become the superuser, issue the *kernel*(1M) command with the protocol as an argument (either **tcp** or **xns**), and reboot the system. Refer to the *GL2-W2.5 Release Notes User's Guide* or *GL2-W3.5 Release Notes* for this procedure. Silicon Graphics, Inc., provides as an option the software for both the VAX host and the IRIS in the XNS environment. In TCP/IP, the software for the IRIS is also available from Silicon Graphics, Inc., as the standard in releases GL2-W2.5 and GL2-W3.5 or later; the software for the VAX is available from the Wollengong Group, of Palo Alto, California. For more information about TCP/IP and XNS, refer to Chapter 2.

## 3.3 IBM Running VM/CMS and OS/MVS

IRIS terminals and workstations can communicate with an IBM host that is running either the VM/CMS or OS/MVS operating system. Silicon Graphics, Inc., provides the Geometry Link, which consists of a 3270 interface card for installation in your IRIS workstation, a 3278/79 emulation program, and host files. You can download applications from the IBM to the IRIS via the 3278/79 emulation program, and you can transfer files between the IRIS and the IBM using the Up Down Transfer (UDT) program. Refer to the *IBM Terminal Emulation* guide from Silicon Graphics, Inc., for complete information on IBM host capabilities with the IRIS. Following are the requirements for operating the IRIS in an IBM environment:

- The IRIS workstation must be running software release GL2-W2.4 (2000 series) or GL2-W3.4 (3000 series) or later.
- The IRIS terminal must be running software release GL2-T2.4 (2000 series) or GL2-T3.4 (3000 series) or later.
- You must install the Silicon Graphics, Inc., 3270 interface card in your terminal or workstation to run the IBM terminal emulation software.
- Before you can use the 3278/79 terminal emulation option, the host files must be installed in the IBM host.

## 3.4 Remote Graphics Library

Silicon Graphics, Inc., provides a Remote Graphics Library (RGL) that resides on a host system (IRIS workstation, VAX, or IBM). RGL programs allow communications of graphics across the network. In order for a terminal to access RGL on a host, the host must also have an IRIS-compatible communications option. For successful communication with RGL, the standard host communications rules apply: communications with a DEC VAX host are with the TCP/IP or XNS protocols; communications with an IBM host are with the Geometry Link. The RGL can run on the following hosts:

- IRIS workstation (C only)
- DEC VAX running UNIX (FORTAN only)
- DEC VAX running VMS (C or FORTAN)

For instructions on installing RGL software on a DEC host and for troubleshooting information, see the *VMS FORTRAN Remote Graphics Library Series 2000/3000* manual. Comparable information for IBM hosts can be found in the *IBM Terminal Emulation* guide.

# 4. Network File System

Network File System (NFS) is a software system developed by Sun Microsystems, Inc., that enables the IRIS to share file systems over the network. It allows a client system (terminal or workstation) to access files residing on another host system on the network. Both systems must be running NFS. The IRIS must have at least four megabytes of available memory. When using NFS, a client mounts file systems from an NFS server.

Silicon Graphics, Inc., has adapted Sun Microsystems, Inc., 3.0 release of NFS as an option for use on a Series 3000 IRIS and on a Series 2000 IRIS with the Turbo option. The IRIS must be running TCP/IP and GL2-W3.5 release software or later. To run NFS, you must become the superuser, issue the *kernel*(1M) command with **nfs** as an argument, and reboot the system.

This section discusses the NFS service, including both a description of how to mount a file system and YP, the NFS lookup service. For complete information on installing and operating NFS, refer to Appendix C, which contains specifications for the *External Data Representation Protocol* and *Remote Procedure Call Protocol*.

## 4.1 Mounting a Remote File System

To work with files from an NFS server, a client must request those files via either a *hard mount* (the default) or *soft mount*. If they are successful, both types of requests effect the file access or operation. A hard mount continues to call the server until it responds. The client waits indefinitely for a response to its request, even if the server is slow or is not functioning. With a soft mount, the client issues the request only a certain number of times, then stops and sends and error message to the console. Before you can mount a file system, your terminal or workstation must be listed as a client in the */etc/export* file on the NFS server.

To issue a mount command, you must first become the superuser. The syntax of the *mount*(1M) command is:

**mount** *server_name:/file_system /mount_point*

The *server_name* is the name of the NFS server, *file_system* is the name of the file system on the NFS server you want to mount, and *mount_point* is the path name on the client where you want the file system to reside. To check that the mounting procedure has been successful, issue the *df*(1) or *mount*(1M) command without an argument. These commands list all the currently mounted file systems on your window.

Typically, you automatically mount frequently used file systems at startup by placing entries for them in the file */etc/fstab* (see *fstab*(4)).

## 4.2 The YP Service

The YP (Yellow Pages) service is the distributed network administrative database for NFS. YP is transparent to the user. It is an option of NFS; it can be disabled by the system administrator. Following is a list of YP features:

- YP is a distributed system. The database is fully replicated at several sites, each of which runs a server process for the database. These sites are known as YP servers. During normal operation, any server process can answer a client request; the answer is the same throughout the network. This allows multiple servers per network, and gives the YP service a high degree of availability and reliability.
- YP is a lookup service. It maintains a set of databases that may be queried. For example, a client may ask for the value associated with a particular key within a database; or the client may request that YP enumerate every key-value pair within a database.

Refer to the *NFS User's Guide* for more information.

# 5. The UUCP Program

UNIX-to-UNIX-Copy (UUCP) is a UNIX utility that enables communication between your IRIS and remote systems. UUCP is the group name of the batch programs that create a dial-up UNIX network. Through UUCP, you can transfer files between local and remote systems, execute remote programs, and send mail between local and remote systems. Because the UUCP network is a batch network, requests for data file transfers and remote execution are spooled by the daemon *uucico*, and transmission does not occur immediately.

## 5.1 Network Requirements

Before you can take advantage of the UUCP utility, you must make sure that the network is properly set up.

The network hardware must be configured to communicate via one of the following:

- Serial line (direct link)
- XNS Ethernet
- Modem running at 1200, 2400, 4800, or 9600 baud

Refer to the *IRIS Series 3000 Owner's Guide* and *IRIS Workstation Guide, Series 2000*, for information on network configuration, or check with your system administrator.

The software required to run the UUCP utility consists of these user programs:

*uucp*(1C) Copies large text files between systems on the local UUCP network. The *uucp* command is one of the programs within UUCP and should be entered in lower-case letters.

*cu*(1) Logs in directly to a remote UNIX system (and to some non-UNIX systems) and executes commands interactively on that system.

*uuto*(1C) Automatically delivers a local source file to the public directory on the remote system and notifies the recipient by mail when the file arrives.

*uupick*(1C) Retrieves files from *usr/spool/uucppublic* after they are transferred using the *uuto* program.

*uux*(1C) Executes commands on a remote system.

## 5.2 UUCP Configuration Files

Running UUCP requires tailoring some UUCP configuration files to your particular network. The UUCP configuration files are in the directories */usr/lib/uucp* and */usr/spool/uucp*. Below is a list of the main UUCP configuration files with descriptions of their functions. A complete set of configuration files can be found in the references listed in Appendix B.

In addition to the UUCP files, two UNIX system files need to be reconfigured for use with UUCP. They are described at the end of this section. For instructions on configuring these, see the *IRIS Series 3000 Owner's Guide* or *IRIS Workstation Guide, Series 2000*.

The following UUCP configuration files are in the directory */usr/lib/uucp*:

*L-devices* Sets the line speeds for the ports used by UUCP. The file contains a series of one-line entries, each of which lists a device, how it is connected, and at what speed it is communicating. For example:

```
DIR tty06 0 4800
```

This entry specifies that the device *tty06* is directly connected and that it is used at 4800 baud.

*L-dialcodes* Contains the dial-code abbreviations used by the */usr/lib/uucp/L.sys* file. The entries are in the form:

*abb dial-sequence*

*abb* is the abbreviation of a location, and *dial-sequence* is the dial sequence associated with the location. For example:

```
sf 415
```

This entry sends the sequence 415 to the dial unit.

*L.cmds* Lists the commands on the local system that can be executed by a remote system through UUCP.

*L.sys* Contains information (e.g., speed of communication) about sites that *uucp*(1) can call.

*USERFILE* Determines the point of access on the system. This can be set to the root directory, to a user, or to a point in a path name of a user. Any files under this point can be accessed by UUCP running as one of the users listed in *USERFILE*.

The following UUCP configuration files are in the directory */usr/spool/uucp*:

*LOGFILE* Shows the UUCP process as it takes place.

*LCK..machinename* Prevent the ports used for UUCP from being interrupted while remote commands are executed.

The following files (with their path names) are UNIX system files that must be configured to enable UUCP communication:

*/etc/passwd* Contains the UUCP user names and passwords of remote systems. All user names and passwords must be defined in this file to allow remote systems to call your system.

*/etc/inittab* Contains a series of lines with *getty* commands for identifying specific serial lines.

## 5.3 Public Directory

UUCP provides the directory */usr/spool/uucppublic* as a convenient intermediate stop for storing files being transferred between remote systems. This directory has general-access privilege; that is, any UUCP user can transfer files from it. Because it is public, if you are transferring files that contain confidential material, you may want to store them in */usr/spool/uucppublic* only briefly, and remove them from the directory as soon as the the recipient notifies you of their receipt. The files in */usr/spool/uucppublic* should be purged regularly by the UUCP administrator.

## 5.4 UUCP Conventions

UUCP requires that you follow certain conventions when naming addresses and paths.

### 5.4.1 Network Addresses

A system's address is a unique name that identifies the system. To function within UUCP, you need to know your system's network address as well as the remote system's address. To find your system's network address, type:

```
uuname -l
```

To find out a remote system's network address, type:

```
uuname
```

A list of all the system addresses appears on your console. The list resides in the file */usr/lib/uucp/L.sys*. Only systems listed in this file can be accessed by UUCP. To add remote system names to the list, you must edit */usr/lib/uucp/L.sys*.

### 5.4.2 Path Names

This section lists the conventions required by UUCP to name paths. They can be used to name either source or destination paths.

- The UUCP notation for a remote path name is:

  *system!pathname* (Bourne shell)
  *system**!pathname* (C shell)

  Note that if you use the C shell, you must override the meaning of an exclamation point by preceding it with a backslash (\). The following examples of UUCP commands use the C-shell notation for remote commands and paths. In this syntax, the *system* is a system that *uucp*(1) knows. The *pathname* part of the command may be either a full path name or a directory name. For example:

  ```
  uucp xnode\!/usr/you/file
  ```
  *or*
  ```
  uucp xnode\!/usr/you/directory
  ```

  In the example above, */usr/you/file* is the entire path name for the file name *file*; */usr/you/directory* is the path name for a directory.

- The login directory on a remote system may be specified by using the tilde (~) character. The combination *~user* references the login directory of a user on the remote system. For example, type:

```
uucp xnode\!~adm/file
```

If the login directory for user *adm* on the remote system is */usr/sys/adm*, the command above is interpreted by UUCP as:

```
uucp xnode\!/usr/sys/adm/file
```

## 5.5 Transferring Files

The *uucp* command with optional arguments can be used to transfer files between local systems and between systems on different networks.

- To transfer a file from a local system to a remote system, issue the *uucp* command with the following syntax:

```
uucp sourcefile destinationfile
```

- To transfer a file into a remote system's directory use the following syntax:

```
uucp file1 remotesys\!/tmp/file1
```

The command above transfers *file1* to another system named *remotesys* into directory */tmp*.

- To fetch a file from a remote system to a local one, type:

```
uucp remotesys\!file1 dir1
```

This command transfers the file named *file1* from the system *remotesys* into the local directory *dir1*.

## 5.6 Forwarding Files

Files that are transferred from one system to another may need to be forwarded through intermediate nodes on the network. The file forwarding feature of UUCP uses a variation of the exclamation point (!) notation to describe the path to be taken to reach files. The syntax for forwarding files has three fields: the command *uucp*, the file name, and the name of each system in the path followed by an exclamation point. For example, a user on system *A* wants to transmit a file to destination *E*. Because systems *B*, *C*, and *D* link systems *A* and *E*, the request must be sent through systems *B*, *C*, and *D* before reaching system *E*. Specify the transfer as follows:

```
uucp file B\!C\!D\!E\!~you/uucppublic/file
```

The file named *file* is sent through *B\!C\!D* and reaches system *E*. In system *E*, the file is sent through the login directory (~) to the *you* directory. Note that the destination must be specified as the public area */usr/spool/uucppublic*.

Fetching a file from another system using intermediate nodes is done in a similar way. For example:

```
uucp B\!C\!D\!E\!~/you/file filex
```

This command fetches *file* from system *E* and renames it *filex* on the local system. The forwarding prefix is the path from the local system, not from the remote system.

The forwarding feature can also be used with remote execution. For example:

```
uux xnode\!uucp ynodepnode!/usr/spool/uucppublic/file filex
```

This command sends a request to *xnode* to execute the *uucp* command to copy a file from *pnode* to *filex* on *xnode*.

## 5.7 Executing Commands on a Remote System

You can use the *uux* command to execute commands on a remote system. *uux* gathers files from various systems, executes a command on the remote system, then sends standard output to a file on the local system. The *uux* command is limited to those commands allowed by the remote system. The remote system's */usr/lib/uucp/L.cmds* file contains the list of commands you can execute on that system. Invoking *uux* queues the request for remote execution of the command. The syntax of *uux* is as follows:

```
uux [option] system\!command
```

The *command* field consists of one or more commands allowed on the remote system, and can include the following special characters and operators. If you use special characters in a command string, enclose them or the entire string in double quotes.

| *Valid* | *Not Valid* |
|---|---|
| < | << |
| > | >> |
| ; | * |
| \| | [] |
| | ? |

Note that, for security reasons, many installations limit the list of commands that can be executed on behalf of an incoming request from *uux*. Many sites permit little more than the receipt of mail. See the *uux*(1C) manual page.

Following is a sample remote execution command:

```
uux "vancouver\!ls > groucho\!~harry/ls.vancouver"
```

This command string consists of two commands: the first executes the *ls* command on the remote system *vancouver*. The second command directs the output of the *ls* command on *vancouver* to a temporary file that *uucp* transfers to user *harry* on system *groucho*.

## 5.8 Sending Mail

UUCP offers a remote message mailing facility. To mail a message to a user on another system, issue the *mail* command with the following syntax:

**mail** *system\!user*

*[message text]*

CTRL-D

CTRL-D ends the mail message and begins the mail transmission.

The variable *system* is the name of the remote system and *user* is the address on the remote system to which the message is sent. See Chapter 6 for more information on mail.

## 5.9 Transferring Files with cu

The *cu* command allows you to transfer files between systems running UNIX and those running non-UNIX operating systems. Although other interfaces are possible, Silicon Graphics, Inc., has successfully tested file transfers between the IRIS and a DEC VAX running the VMS operating system.

To use *cu* effectively, you need adequate permission to read and write files. At the local system prompt, type:

**chmod 666** *filename*

*filename* is the name of the file you want to transfer. Type:

```
ls -l /dev/ttyd1
```

The system responds with a message similar to:

```
crw-rw-rw- 1 larry  40,  49 Oct 8 10:26 /dev/ttyd1
```

Type:

```
cat /usr/lib/uucp/L-devices
```

The system responds with:

```
DIR ttyd1          ttyd1          4800
DIR ttyd2          ttyd2          2400
DIR ttyd3          ttyd3          1200
DIR ttyd4          ttyd4          300
DIR xns            xns            xns
```

This checks the the port's connection and speed. To connect to a serial port at 9600 baud, type:

```
cu -lttyd1 -s9600
```

To connect to a modem, refer to the modem manufacturer's instructions.

Sections 5.9.1 and 5.9.2 explain the procedure for transferring files between an IRIS and a VAX running VMS using the *cu* command.

### 5.9.1 IRIS to VAX Transfers

This procedure transfers files from the IRIS to a VAX at 9600 baud.

1. After logging in to your IRIS, type:

   ```
   cu -lttyd1 -s9600 dir
   ```

2. Log in to the VAX from your IRIS. At the dollar sign ($) prompt from the VAX, type:

   ```
   set term/hostsync/readsync/ttsync/noecho
   ```

   The *noecho* portion of the command makes the next line you type invisible.

3. Give the file you want to copy from the IRIS a file name on the VAX. The syntax of the command is:

   **create** *file_to_be_dumped_to*

For example:

```
create xferfile
```

4. The syntax of the transfer command is:

```
~$tr "\012" "\015" < file_on_IRIS
```

In the example above, it would look like:

```
~$tr "\012" "\015" < xferfile
```

This line is visible because you have broken through to the UNIX shell on the IRIS.

5. When you see the dollar sign prompt from the VAX, type:

CTRL-Z

The CTRL-Z is invisible.

6. To exit *cu*, type

```
set term/echo
~.
```

## 5.9.2 VAX to IRIS Transfers

This procedure transfers files from a DEC VAX to an IRIS at 9600 baud.

1. After logging in to the IRIS, type:

```
cu -lttyd1 -s9600 dir
```

2. Log in to the VAX. When you see the dollar sign ($) prompt from the VAX, type:

```
write sys$output "~>:file_on_IRIS"
```

The next two lines you enter will be invisible.

3. Type:

```
type file_to_be_sent
write sys$output "~>"
```

4. To exit *cu*, type:

```
~.
```

You can create a command file to run on the VAX to perform VAX to IRIS transfers more easily. This is a sample command file:

```
$! This script sends a file from VMS/VAX
$! to the IRIS  via cu. To use,
$! type  @sendit file-on-IRIS file-to-be-sent
$!
$  write sys$output "~>:''p1'"
$  type 'p2'
$  write sys$output "~>"
$  exit
```

# 6. The Mail System

The mail system is a group of programs that let you send messages to other users on the network and to receive messages from them. This section provides an in-depth discussion of the mail system, including:

- Sending mail
- Address aliasing
- UUCP and sendmail
- Mail administration

You can send mail on the IRIS through either UUCP or TCP/IP on an IRIS host running GL2-W2.5, GL2-W3.5, or a later release. UUCP is used on IRISes running XNS or using serial lines or modems. See Chapter 5 and Section 6.5 for information on UUCP.

Silicon Graphics, Inc., uses System V.0 */bin/mail* and 4.3bsd *sendmail* and *Mail* for its mail implementation.

The following software must be in place to run mail:

- */usr/lib/sendmail.cf*
- */usr/lib/sendmail*
- */usr/bin/Mail*
- */bin/mail*
- */bin/rmail*

## 6.1 Mail System Hierarchy

The mail system programs can be divided into four functional categories:

| | |
|---|---|
| User interfaces | These programs allow composition and perusal of the mail text. They provide a user interface that supports the creation of new messages and the reading and removal or archiving of received messages. Examples of this level of the mail system are *Mail*(1) and *mail*(1). |
| Mail routing | The *sendmail*(1) program calls the mail delivery or mail transmission program. Routing messages in UUCP is explicit; it must be done by the user. |
| Mail delivery | These programs are responsible for depositing mail into a data file for later perusal by a user or another program. |
| Mail transmission | Mail transmission is needed when the destination for the mail resides on a remote host. Examples of this level of the mail system are UUCP, which uses its own protocols and runs over XNS and serial lines, and *sendmail*, which uses the Standard Mail Transmission Protocol (SMTP) and runs with TCP/IP. In all cases, the mail transmission process has a counterpart, the mail reception process. Both processes reside in the same program. |

After you compose a message using an available user interface, the message is sent to a mail transmission or routing program.

## 6.2 Examples Using Mail

Sections 6.2.1 and 6.2.2 illustrate sample mail messages. The first shows a message sent without routing. The second message is routed through UUCP. Note that in both examples the syntax of the *Mail* command uses the C-shell notation:

```
host\!user
```

The command could be issued as well with the Bourne shell notation:

```
host!user
```

The symbol "%" prompts for user interaction with the system. All messages end by typing CTRL-D on a separate line.

### 6.2.1 Mail without Routing

The following is an example of a common host interconnection. Hosts *one, two* and *three* are on an Ethernet connection and use SMTP (the Standard Mail Transmission Protocol). Host *four* is connected to host *one* via a serial line, and can communicate only via UUCP. Figure 6-1 shows four users: *fred* on host *one, barney* on host *two, wally* on host *three,* and *june* on host *four.*

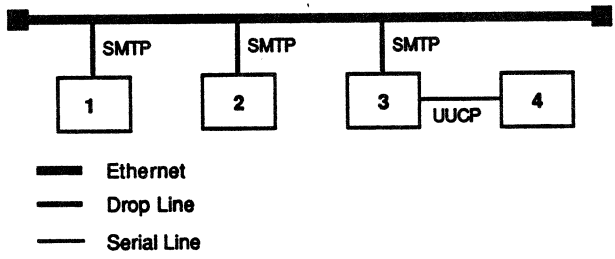



Figure 6-1: Map of Users and Hosts

To send mail, type *Mail* and the address at the system prompt (%); the prompt does not appear again until you exit from *Mail.*

If user *fred* wants to send mail to *barney, fred* types:

```
% Mail two\!barney
Subject: Bedrock construction

Hey Barney
Let's go fishing!
CTRL-D
%
```

When *Mail* is finished composing the message, it calls the *sendmail* program to figure out how to send the mail to *barney.* The choice of calling *sendmail* is part of the *Mail* program. *sendmail* figures out how to send the mail to *barney* and which of its resources can perform the transmission. In this simple case, because hosts *one* and *two* are on the same network, and because they both use SMTP, *sendmail* uses TCP/IP directly to send the mail. A transmission program that uses SMTP is built into *sendmail.*

On host *two,* it is assumed that a daemon exists that uses SMTP and that the SMTP transmission program on host *one* will connect to it. Once this occurs,

the mail is transmitted to host *two* and is ready for delivery. The *sendmail* program runs on host *two* and receives the SMTP message. After removing host *two*'s address from the destination address, *sendmail* runs through an algorithm to resolve the new destination address.

The process used to resolve this address is approximately the same as that used in the original sending of the address, when *fred* sent the original message. For this example, the resolved address is *barney*. The local mail delivery program, */bin/mail -d*, is called to deposit the mail into *barney*'s mailbox.

## 6.2.2 Mail with Routing

In this example, *barney* on host *two* wants to send mail to *june* on host *four:*

```
% Mail four\!june
Subject: Your tv show

It's great. Let's have lunch Tuesday.

Sincerely, Barney Rubble
CTRL-D
%
```

*Mail* composes the message and hands it to *sendmail* to route. *sendmail*, via its configuration file on host *two* determines that it cannot send the mail directly to host *four*. Instead, the mail must be routed through host *one*, because host *one* has the only direct connection to host *four*. *sendmail* rewrites the mail address to *one!four!june*, then sends the message to the SMTP server on host *one*. Host *one* strips off its address, resulting in the address *four!june*.

Not all mailers use the same address format. UUCP does not use the same format as *sendmail*. To cope with this, the *sendmail.cf* configuration file rewrites addresses for the mailer. Once the *four!june* address is received at host *one*, the *sendmail.cf* file for host *one* determines that it must use UUCP to send mail to host *four*. Note that this process is transparent to *barney*, the originator of the message.

Once *sendmail* decides that UUCP must be used to send the mail, the data in the *sendmail.cf* file instructs *sendmail* in how to call the UUCP mail transmission program. UUCP then takes the message and mails it to *june* via *rmail* (*rmail* is responsible only for delivering UUCP mail). On host *four*, 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; *june* receives the original message.

## 6.3 The sendmail.cf File

The *sendmail.cf* file identifies and contains information about all systems on the network. Before a system can function with the mail program, the *sendmail.cf* file must be modified to identify the system. The file entry for a system includes the system's name and the transmission protocols it uses. When *sendmail* is invoked to send a message, it uses the rules in *sendmail.cf* to resolve the "To" address to a particular system, user, and mailer (the program that transmits the message).

When modifying the *sendmail.cf* file, keep in mind the following points:

- If a system is in more than one category in the file, the order of protocol choices is important. If both the target systems and the current system are declared within the *sendmail.cf* file to use SMTP, then SMTP is used to deliver the mail to the target. Otherwise, UUCP is used.
- Avoid using upper-case letters for the names of users and hosts. Many systems are case-insensitive, and some automatically convert upper-case letters to lower case.

The *sendmail.cf* file is divided into sections by the type of transmission protocol. In each section, the names of the systems that communicate using that protocol are listed. Following are examples of some of the types of entries in *sendmail.cf*:

- *SMTP systems*

  These systems fall into the "CS" category. To add a system running SMTP, search the file for lines beginning with "CS" and add the system's name to it. For example, if your system named *chaplin* runs SMTP, edit *sendmail.cf* and search for "CS." The search takes you to these lines:

```
# Direct connect smtp hosts
CSgroucho
CShardy
```

  Add *chaplin*, with the prefix "CS," to the list:

```
# Direct connect smtp hosts
CSgroucho
CShardy
CSchaplin
```

  If you do not send mail to certain systems, for example, *groucho* or *hardy*, delete them from this list.

- *Modem systems*

  If you have systems with modems, search for that category. Edit *sendmail.cf* and search for "CL" lines. The search takes you to these lines:

```
# Machines with local modems - **WARNING** these machines WON'T forward
# unknown mail.
CLabbott
CLcostello
```

  Add sytems with modems to this list. Be sure to prefix your entry with "CL."

- *UUCP or XNS systems*

  If you have systems with UUCP, search for those with direct UUCP connections. (For a discussion of UUCP, see Chapter 5.) Also use the "CU" class for systems connected with XNS. The search for "CU" takes you to these lines:

```
# direct UUCP connections
CUabbott
CUharpo
```

  Add any UUCP systems, including the "CU" prefix, to the list.

  All UUCP and XNS systems must also be added to the UNIX system file *L.sys*. Refer to Chapter 5 and Section 6.5 for more information on these files.

The *sendmail.cf* file on each IRIS must be tailored to fit the type of mail transmission, UUCP or SMTP, of each host to which it sends mail. For example, host *one* shown in Figure 6-1 uses SMTP to send mail to hosts *two* and *three*, and uses UUCP to send mail to host *four*. Host *one*'s *sendmail.cf* file is shown below:

```
# Direct connect smtp hosts
CStwo
CSthree
# direct UUCP connections
CUfour
```

## 6.4 Address Aliasing

Address aliasing is a process whereby one address is converted to 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, thus 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. Modify this file to add or delete aliases.

In the following example, the alias:

```
brubble: two\!barney
```

is in */usr/lib/aliases*. *brubble* is what you type. *two\!barney* is the routing followed by sendmail. The message travels through system *two* to user *barney*.

Aliasing is performed only on local addresses. A message can be sent to a remote location via 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 contained in the file. If mail is forwarded, *sendmail* applies the same rules that it applied to the original recipient address, and restarts the routing process.

## 6.5 UUCP and Sendmail

The *sendmail* program uses UUCP to deliver mail to any system that can be reached through UUCP (specified as class "CU" in the *sendmail.cf* file; see Section 6.3) or to an unknown system.

UUCP has its own routing system. The routing of the message with a UUCP destination must be done explicitly once the message crosses outside the name space of *sendmail*. If UUCP cannot deliver mail (if either the destination host or user is unknown), the mail is returned with an error message that points out the part of the address that is in error. For example, if you send mail to *moose\!woods* and *woods* is unknown, the error message would be similar to this and would be followed by the text of the unsent message:

```
bad system name woods
----- Transcript of session follows ----
uux failed.  code 101
554 woods\!moose unknown mailer error

----- Unsent message follows -----
```

In order for UUCP to function in the mail program, all UUCP and XNS systems on the network must be listed in the UNIX system file *L.sys*. You must become the superuser to edit this file. Search the *L.sys* file for this line:

```
works2 Any xns xns xns "" \r\c ogin:--ogin: uucpco assword: secret
```

In the example below, the above line has been modified to reflect the new XNS system named *pretzel*, with the UUCP login *uucpws* and the password *mustard*:

```
pretzel Any xns xns xns "" \r\c ogin:--ogin: uucpws assword: mustard
```

The password in *L.sys* for the *uucpws* account on the system *pretzel* must be the same as that in *pretzel*'s */etc/passwd* file. Also, a line in your */etc/passwd* must begin with *uucpws* and should look similar to the following line:

```
uucpws:xtpVRtjwxL03Q:3:5:UUCP Login Account:/usr/spool/uucppublic:/usr/lib/uucp/uu
```

# Appendix A: Communications Options on the IRIS

The table below displays communication options that run on the IRIS Series 2000 and 3000. The IRIS 2400 Turbo and 2500 Turbo are included with the IRIS Series 3000, because they run the GL2-W3.5 software release.

| Communications Option / Host | TCP/IP, IBM, Serial line, IEEE 488 | NFS |
|---|---|---|
| IRIS Series 2000 running GL2-W2.5 release software | **Available** | **Not Available** |
| IRIS Series 2000/3000 running GL2-W3.5 release software | **Available** | **Available** |

**Communication Options on the IRIS**

The table below displays the host operating systems with which the IRIS can communicate.

| Hosts / IRIS | DEC VAX running UNIX | DEC VAX running VMS | IBM running either OS/MVS or VM/CMS | IRIS workstation |
|---|---|---|---|---|
| IRIS Series 2000 running GL2-W2.5 release software | TCP/IP XNS Serial line | TCP/IP XNS Serial line | Geometry Link | TCP/IP, Serial line, or XNS |
| IRIS Series 2000/3000 running GL2-W3.5 release software | TCP/IP XNS Serial line | TCP/IP XNS Serial line | Geometry Link | TCP/IP, NFS, Serial line, or XNS |

**Host Communications on the IRIS**

# Appendix B: Bibliography

This appendix is designed to direct you to further reading about topics covered in the *IRIS Communications Guide*. References published by Silicon Graphics, Inc., are followed by the abbreviation SGI.

## B.1 IRIS Manuals

- *IRIS Series 3000 Owner's Guide*, SGI
- *IRIS Workstation Guide, Series 2000*, SGI
- *IRIS Terminal Guide, Series 2000*, SGI
- *Getting Started with Your IRIS Workstation*, SGI

## B.2 Protocols

- *TCP/IP User's Guide*, SGI
- *Defense Data Network Protocol Handbook*
- "Installing Software Updates" in the current release notes, SGI
- *UNIX Programmer's Manual*, SGI

### B.2.1 XNS

- *IRIS XNS User's Guide*, SGI

## B.3 Host Communications

- *IBM Terminal Emulation*, SGI

### B.3.1 DEC VAX Running VMS

- *VMS FORTRAN Remote Graphics Library, Series 2000/3000*, SGI
- *VMS XNS Series 2000/3000 Software Guide*, SGI

## B.4 NFS

The documents described below are published by Sun Microsystems, Inc. The *Remote Procedure Call Protocol Specification* and *External Data Representation Protocol Specification* are in the Appendix C of this manual.

## B.5 UUCP

- "UUCP," in the *UNIX Programmer's Manual, Volume I*, SGI
- "UUCP Tutorial," in the *UNIX Programmer's Manual, Volume I*, SGI
- Todino, Grace, *Using UUCP and Usenet*, Nutshell Handbooks, a trademark of O'Reilly and Associates, Inc.
- Todino, Grace, *Managing UUCP and Usenet*, Nutshell Handbooks, a trademark of O'Reilly and Associates, Inc.

## B.6 Mail

Appendices D, E, F, and G contain documents on mail.

# Appendix C: NFS Reference Materials

**Introduction to XDR and RPC**

The libraries and include files required to use the routines documented in the the *External Data Representation Protocol Specification* and *Remote Procedure Call Protocol Specifications* from Sun Microsystems, Inc., are supplied with the NFS software option.

The include files that are referenced in these documents reside in the */usr/include/rpc* directory.

In order to link a program that calls the routines documented in the *RPC Protocol Specifications* and the *XDR Protocol Specifications*, it is necessary to include two libraries in the load (*ld*) command: Sun Microsystems, Inc.'s RPC support library *(/usr/lib/libsun.a)* and the Berkeley 4.3 compatibility library *(/usr/lib/libbsd.a)*. The BSD library must be included because the RPC functions call procedures that are defined only in the BSD library. The easiest way to include these libraries is by using a compiler command similar to the following:

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

Note that the libraries must be included in the order shown. Refer to the *intro*(3) section of the *UNIX Programmer's Manual* for more information.

# External Data Representation Protocol Specification

## 1. Introduction

This manual describes library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. The eXternal Data Representation (XDR) standard 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 manual contains a description of XDR library routines, a guide to accessing currently available XDR streams, information on defining new streams and data types, and a formal definition of the XDR standard. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) only need the information in sections 2 and 3 of this document. Programmers wishing to implement RPC and XDR on new machines will need the information in sections 4 through 6. Advanced topics, not necessary for all implementations, are covered in section 7.

On Sun systems, 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 `libc.a` contains all the XDR routines, compile as normal.

```
cc program.c
```

### 1.1. Justification

Consider the following two programs, `writer`:

```
#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);
                }
        }
}
```

and `reader`:

```
#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");
}
```

The two programs appear to be portable, because (a) they pass `lint` checking, and (b) they exhibit the same behavior when executed on two different hardware architectures, a Sun and a VAX.

Piping the output of the `writer` program to the `reader` program gives identical results on a Sun or a VAX. ‡

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%

vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

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`. Here are the results if the first produces data on a Sun, and the second consumes data on a VAX.

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
sun%
```

Identical results can be obtained by executing `writer` on the VAX and `reader` on the Sun. These results occur because the byte ordering of long integers differs between the VAX and the Sun, 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. Here are 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);
                }
        }
}
```

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");
}
```

The new programs were executed on a Sun, on a VAX, and from a Sun to a VAX; the results are shown below.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%

vax% writer | reader
0 1 2 3 4 5 6 7
vax%

sun% writer | rsh vax reader
0 1 2 3 4 5 6 7
sun%
```

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.

### 1.2. The XDR Library

The XDR library solves data portability problems. It allows you to write and read arbitrary C constructs in a consistent, specified, well-documented manner. Thus, it makes 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.

Let's 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 our example, data is manipulated using standard I/O routines, so we 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 clients never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the clients.

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:

```
xdr_xxx(xdrs, fp)
        XDR *xdrs;
        xxx *fp;
{
}
```

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, you can obtain the direction of the XDR operation. See section 3.7 for details.

Let's 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

#define enum_t int      /* enum_t used for generic enums */
```

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.

## 2. 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>`.

### 2.1. 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 six primitives are:

```
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.

### 2.2. 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. All 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.

### 2.3. 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.

### 2.4. 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 */
```

### 2.5. 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`.

#### 2.5.1. 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

## 2. 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>`.

### 2.1. 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 six primitives are:

```
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.

### 2.2. 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. All 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.

### 2.3. 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.

### 2.4. 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 */
```

### 2.5. 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`.

#### 2.5.1. 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 exceeds `maxlength`, and `TRUE` if it doesn't.

The behavior of `xdr_string()` is similar to the behavior of other routines discussed in this section. The direction `XDR_ENCODE` is easiest to understand. The parameter `sp` points to a string of a certain length; if it 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.

### 2.5.2. 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.

### 2.5.3. Arrays

The XDR library package provides a primitive for handling arrays of arbitrary elements. The `xdr_bytes()` routine treats a subset of generic arrays, in which the size of array elements is known to be 1, and the external description of each element is built-in. The generic array primitive, `xdr_array()` requires parameters identical to those of `xdr_bytes()` plus two more: the size of array elements, and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array.

```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsiz, xdr_element)
        XDR *xdrs;
        char **ap;
        u_int *lp;
        u_int maxlength;
        u_int elementsiz;
        bool_t (*xdr_element)();
```

The parameter `ap` is the address of the pointer to the array. If `*ap` is `NULL` when the array is being deserialized, XDR allocates an array of the appropriate size and sets `*ap` to that array. The element count of the array is obtained from `*lp` when the array is serialized; `*lp` is set to the array length when the array is deserialized. The parameter `maxlength` is the maximum number of elements that the array is allowed to have; `elementsiz` is the byte size of each element of the array (the C function `sizeof()` can be used to obtain this value). The routine `xdr_element` is called to serialize, deserialize, or free each element of the array.

#### 2.5.3.1. Examples

Before defining more constructed data types, it is appropriate to present three examples.

*Example A*

A user on a networked machine can be identified by (a) the machine name, such as `krypton`: see *gethostname* (3); (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 < 256 chars */
#define NGRPS 20        /* user can't be in > 20 groups */

bool_t
xdr_netuser(xdrs, nup)
        XDR *xdrs;
        struct netuser *nup;
{
        return(xdr_string(xdrs, &nup->nu_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:

```
struct cmd {
        u_int c_argc;
        char **c_argv;
};
#define ALEN 1000       /* args cannot be > 1000 chars */
#define NARGC 100       /* commands cannot have > 100 args */

struct history {
        u_int h_len;
        struct cmd *h_cmds;
};
#define NCMDS 75  /* history is no more than 75 commands */

bool_t
xdr_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. Let's continue with more constructed data types.

### 2.5.4. 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.

### 2.5.5. Fixed Sized 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 fashion:

```
#define NLEN 255        /* machine names must be < 256 chars */
#define NGRPS 20        /* user can't belong to > 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);
        for (i = 0; i < NGRPS; i++) {
                if (!xdr_int(xdrs, &nup->nu_gids[i]))
                        return(FALSE);
        }
        return(TRUE);
}
```

Exercise: Rewrite example A so that it uses varying-length arrays and so that the `netuser` structure contains the actual `nu_gids` array body as in the example above.

### 2.5.6. 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 order 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 in Section 2; `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.

### 2.5.7. Pointers

In C it is often convenient to put pointers to another structure within a structure. The primitive `xdr_reference()` 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_struc()` to describe structures within structures, because pointers are always sufficient.

Exercise: Implement `xdr_reference()` using `xdr_array()`. Warning: `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 construct is:

```
struct pgn {
        char *name;
        struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
        XDR *xdrs;
        struct pgn *pp;
{
        if (xdr_string(xdrs, &pp->name, NLEN) &&
          xdr_reference(xdrs, &pp->gnp,
          sizeof(struct gnumbers), xdr_gnumbers))
                return(TRUE);
        return(FALSE);
}
```

#### 2.5.7.1. 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 UNIX, a core dump for debugging.

It is the explicit responsibility of the programmer to expand non-dereferenceable pointers into their specific semantics. This usually involves describing data with a two-armed discriminated union. One arm is used when the pointer is valid; the other is used when the pointer is invalid (NULL). (Section 7 has an example (linked lists encoding) that deals with invalid pointer interpretation.

Exercise: After reading Section 7, 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. The XDR library does not provide such a primitive because it does not want to give the illusion that pointers have meaning in the external world.

### 2.6. 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. Warning: 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.

### 2.7. 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 Section 7 demonstrates the usefulness of the `xdrs->x_op` field.

### 3. 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 UNIX files, and memory. Section 5 documents the XDR object and how to make new XDR streams when they are required.

#### 3.1. 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.

#### 3.2. 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.

#### 3.3. 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 part of rpc */

xdrrec_create(xdrs,
  sendsize, recvsize, iohandle, readproc, writeproc)
        XDR *xdrs;
        u_int sendsize, recvsize;
        char *iohandle;
        int (*readproc)(), (*writeproc)();
```

The routine `xdrrec_create()` provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX files.

The parameter `xdrs` is similar to the corresponding parameter described 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 needs to be filled or flushed, the routine `readproc` or `writeproc` is called, respectively. The usage and behavior of these routines are similar to the UNIX system calls `read()` and `write()`. However, the first parameter to each of these routines is the opaque parameter `iohandle`. The other two parameters `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 appendix 1. 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, then the routine `xdrrec_eof()` returns `TRUE`. That is not to say that there is no more data in the underlying file descriptor.

## 4. XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

### 4.1. 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.

Macros for accessing operations `x_getpostn()`, `x_setpostn()`, and `x_destroy()` were defined in Section 3.6. 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 UNIX primitives `htonl()` and `ntohl()` can be helpful in accomplishing this. Section 6 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.

## 5. XDR Standard

This section defines the external data representation standard. The standard is independent of languages, operating systems and hardware architectures. Once data is shared among machines, it should not matter that the data was produced on a Sun, 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.

The external data representation standard depends on the assumption that bytes (or octets) are portable. A byte is defined to be eight 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 "little endian" style. Both Sun and VAX hardware implementations adhere to the standard.

The XDR standard also suggests a language used to describe data. The language is a bastardized C; it is a data description language, not a programming language. (The Xerox Courier Standard uses bastardized Mesa as its data description language.)

### 5.1. 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, where (n mod 4) = 0. The bytes are read or written to some byte stream such that byte *m* always precedes byte *m*+1.

### 5.2. 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`.

### 5.3. 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`.

### 5.4. 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:

```
typedef enum { name = value, .... } type-name;
```

For example the three colors red, yellow and blue could be described by an enumerated type:

```
typedef enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

### 5.5. Booleans

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Boolean is an enumeration with the following form:

```
typedef enum { FALSE = 0, TRUE = 1 } boolean;
```

### 5.6. Hyper Integer and Hyper Unsigned

The standard also defines 64-bit (8-byte) numbers called `"hyper integer"` and `"hyper unsigned"`. Their representations are the obvious extensions of the integer and unsigned defined above. The most and least significant bytes are 0 and 7, respectively.

### 5.7. Floating Point and Double Precision

The standard defines the encoding for the floating point data types `float` (32 bits or 4 bytes) and `double` (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized single- and double-precision floating point numbers. See the IEEE floating point standard for more information. The standard encodes the following three fields, which describe the 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. Floats devote 8 bits to this field, while doubles devote 11 bits. The exponents for float and double are biased by 127 and 1023, respectively.

*F* The fractional part of the number's mantissa, base 2. Floats devote 23 bits to this field, while doubles devote 52 bits.

Therefore, the floating point number is described by:

$$(-1)^S * 2^{E-Bias} * 1.F$$

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.

Doubles have the analogous extensions. The beginning bit (and most significant bit) offsets of *S*, *E*, and *F* are 0, 1, and 12, respectively.

The IEEE specification should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). Under IEEE specifications, the "NaN" (not a number) is system dependent and should not be used.

### 5.8. Opaque Data

At times fixed-sized uninterpreted data needs to be passed among machines. This data is called `opaque` and is described as:

```
typedef opaque type-name[n];
opaque name[n];
```

where `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 (up to 3) zero-valued bytes to make the total byte count of the opaque object a multiple of four.

### 5.9. Counted Byte Strings

The standard defines a string of *n* (numbered 0 through *n*−1) bytes to be the number *n* encoded as `unsigned`, and followed by the *n* bytes of the string. If *n* is not a multiple of four, then the *n* bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count a multiple of four. The data description of strings is as follows:

```
typedef string type-name<N>;
typedef string type-name<>;
string name<N>;
string name<>;
```

Note that the data description language uses angle brackets (< and >) to denote anything that is varying-length (as opposed to square brackets to denote fixed-length sequences of data).

The constant N denotes an upper bound of the number of bytes that a string may contain. If N is not specified, it is assumed to be $2^{32}-1$, the maximum length. The constant N 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, such as:

```
string filename<255>;
```

The XDR specification does not say what the individual bytes of a string represent; this important information is left to higher-level specifications. A reasonable default is to assume that the bytes encode ASCII characters.

### 5.10. Fixed Arrays

The data description for fixed-size arrays of homogeneous elements is as follows:

```
typedef elementtype type-name[n];
elementtype name[n];
```

Fixed-size arrays of elements numbered 0 through $n-1$ are encoded by individually encoding the elements of the array in their natural order, 0 through $n-1$.

### 5.11. Counted Arrays

Counted arrays provide the ability to encode varyiable-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 data description for counted arrays is similar to that of counted strings:

```
typedef elementtype type-name<N>;
typedef elementtype type-name<>;
elementtype name<N>;
elementtype name<>;
```

Again, the constant N specifies the maximum acceptable element count of an array; if N is not specified, it is assumed to be $2^{32}-1$.

### 5.12. Structures

The data description for structures is very similar to that of standard C:

```
typedef struct {
        component-type component-name;
        ...
} type-name;
```

The components of the structure are encoded in the order of their declaration in the structure.

### 5.13. 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 the discriminant is always an enumeration. The component types are called "arms" of the union. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm. The data description for discriminated unions is as follows:

```
typedef union switch (discriminant-type) {
        discriminant-value: arm-type;
        ... .
        default: default-arm-type;
} type-name;
```

The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. Most specifications neither need nor use default arms.

### 5.14. Missing Specifications

The standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. This is not to say that no specification should be attempted.

### 5.15. Library Primitive / XDR Standard Cross Reference

The following table describes the association between the C library primitives discussed in Section 3, and the standard data types defined in this section:

| *Primitives and Data Types* | | |
|---|---|---|
| *C Primitive* | *XDR Type* | *Sections* |
| `xdr_int`<br>`xdr_long`<br>`xdr_short` | `integer` | 3.1, 6.2 |
| `xdr_u_int`<br>`xdr_u_long`<br>`xdr_u_short` | `unsigned` | 3.1, 6.3 |
| - | `hyper integer`<br>`hyper unsigned` | 6.6 |
| `xdr_float` | `float` | 3.2, 6.7 |
| `xdr_double` | `double` | 3.2, 6.7 |
| `xdr_enum` | `enum_t` | 3.3, 6.4 |
| `xdr_bool` | `bool_t` | 3.3, 6.5 |
| `xdr_string`<br>`xdr_bytes` | `string` | 3.5.1, 6.9<br>3.5.2 |
| `xdr_array` | (varying arrays) | 3.5.3, 6.11 |
| - | (fixed arrays) | 3.5.5, 6.10 |
| `xdr_opaque` | `opaque` | 3.5.4, 6.8 |
| `xdr_union` | `union` | 3.5.6, 6.13 |
| `xdr_reference` | - | 3.5.7 |
| - | `struct` | 6.6 |

## 6. Advanced Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the earlier sections, the following examples are written using both the XDR C library routines and the XDR data description language. Section 6 describes the XDR data definition language used below.

### 6.1. Linked Lists

The last example in Section 2 presented a C data structure and its associated XDR routines for a person'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 we wish to implement a linked list of such information. A data structure could be constructed as follows:

```
typedef struct gnnode {
        struct gnumbers gn_numbers;
        struct gnnode *nxt;
};

typedef struct gnnode *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 `nxt` field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the `nxt` 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:

```
struct gnumbers {
        unsigned g_assets;
        unsigned g_liabilities;
};

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 `nxt` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing a set of XDR routines to successfully (de)serialize a linked list of entries can be taken from the XDR description of the pointer-less data. The set consists of the mutually recursive routines `xdr_gnumbers_list`, `xdr_wrap_list`, and `xdr_gnnode`.

```
bool_t
xdr_gnnode(xdrs, gp)
        XDR *xdrs;
        struct gnnode *gp;
{
        return(xdr_gnumbers(xdrs, &(gp->gn_numbers)) &&
                xdr_gnumbers_list(xdrs, &(gp->nxt)) );
}

bool_t
xdr_wrap_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        return(xdr_reference(xdrs, glp, sizeof(struct gnnode),
            xdr_gnnode));
}

struct xdr_discrim choices[2] = {
        /*
         * called if another node needs (de)serializing
         */
        { TRUE, xdr_wrap_list },
        /*
         * called when no more nodes need (de)serializing
         */
        { FALSE, xdr_void }
}

bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        more_data = (*glp != (gnumbers_list)NULL);
        return(xdr_union(xdrs, &more_data, glp, choices, NULL));
}
```

The entry routine is `xdr_gnumbers_list()`; its job is to translate between the boolean value `more_data` and the list pointer values. If there is no more data, the `xdr_union()` primitive calls `xdr_void()` and the recursion is terminated. Otherwise, `xdr_union()` calls `xdr_wrap_list()`, whose job is to dereference the list pointers. The `xdr_gnnode()` routine actually (de)serializes data of the current node of the linked list, and recursively calls `xdr_gnumbers_list()` to handle the

remainder of the list.

You should convince yourself that these routines function correctly in all three directions (XDR_ENCODE, XDR_DECODE, and XDR_FREE) for linked lists of any length (including zero). Note that the boolean more_data is always initialized, but in the XDR_DECODE case it is overwritten by an externally generated value. Also note that the value of the bool_t is lost in the stack. The essence of the value is reflected in the list's pointers.

The unfortunate side effect of (de)serializing a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. The routines are also hard to code (and understand) due to the number and nature of primitives involved (such as xdr_reference, xdr_union, and xdr_void).

The following routine collapses the recursive routines. It also has other optimizations that are discussed below.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        while (TRUE) {
                more_data = (*glp != (gnumbers_list)NULL);
                if (!xdr_bool(xdrs, &more_data))
                        return(FALSE);
                if (!more_data)
                        return(TRUE);  /* we are done */
                if (!xdr_reference(xdrs, glp, sizeof(struct gnnode),
                    xdr_gnumbers))
                        return(FALSE);
                glp = &((*glp)->nxt);
        }
}
```

The claim is that this one routine is easier to code and understand than the three recursive routines above. (It is also buggy, as discussed below.) The parameter glp is treated as the address of the pointer to the head of the remainder of the list to be (de)serialized. Thus, glp is set to the address of the current node's nxt field at the end of the while loop. The discriminated union is implemented in-line; the variable more_data has the same use in this routine as in the routines above. Its value is recomputed and re-(de)serialized each iteration of the loop. Since *glp is a pointer to a node, the pointer is dereferenced using xdr_reference(). Note that the third parameter is truly the size of a node (data values plus nxt pointer), while xdr_gnumbers() only (de)serializes the data values. We can get away with this tricky optimization only because the nxt data comes after all legitimate external data.

The routine is buggy in the XDR_FREE case. The bug is that xdr_reference() will free the node *glp. Upon return the assignment "glp = &((*glp)->nxt)" cannot be guaranteed to work since *glp is no longer a legitimate node. The following is a rewrite that works in all cases. The hard part is to avoid dereferencing a pointer which has not been initialized or which has been freed.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;
        bool_t freeing;
        gnumbers_list *next;  /* the next value of glp */

        freeing = (xdrs->x_op == XDR_FREE);
        while (TRUE) {
                more_data = (*glp != (gnumbers_list)NULL);
                if (!xdr_bool(xdrs, &more_data))
                        return(FALSE);
                if (!more_data)
                        return(TRUE);  /* we are done */
                if (freeing)
                        next = &((*glp)->nxt);
                if (!xdr_reference(xdrs, glp, sizeof(struct gnnode),
                    xdr_gnumbers))
                        return(FALSE);
                glp = (freeing) ? next : &((*glp)->nxt);
        }
}
```

Note that this is the first example in this document that actually inspects the direction of the operation `xdrs->x_op` ). (The claim is that the correct iterative implementation is still easier to understand or code than the recursive implementation. It is certainly more efficient with respect to C stack requirements.

### 6.2. The Record Marking Standard

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 that 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 high-order bit of the header; the length is the 31 low-order bits.

(Note that this record specification is *not* in XDR standard form and cannot be implemented using XDR primitives!)

**Appendix A -- Synopsis of XDR Routines**

`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_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_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 the pointer is cast to `"long *"`. Warning: `xdr_inline()` may return `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. This routine 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_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_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_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 in the address of the union. This routine returns one if it succeeds, zero otherwise.

`xdr_void()`

```
xdr_void()
```

This routine always returns one. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

`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 `fclose()`.

# Remote Procedure Call Protocol Specification

## 7. Introduction

This document specifies a message protocol used in implementing Sun's Remote Procedure Call (RPC) package. The message protocol is specified with the eXternal Data Representation (XDR) language.

This document assumes that the reader is familiar with both RPC and XDR. 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 document.

### 7.1. 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.

### 7.2. 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. That is, the RPC protocol does not explicitly support multi-threading of caller or server processes.

### 7.3. 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.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Some semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, RPC message passing using UDP/IP is unreliable. Thus, if the

caller retransmits call messages after short time-outs, the only thing he can infer from no reply message is that the remote procedure was executed zero or more times (and from a reply message, one or more times). On the other hand, RPC message passing using TCP/IP is reliable. No reply message means that the remote procedure was executed at most once, whereas a reply message means that the remote procedure was exactly once. (Note: At Sun, RPC is currently implemented on top of TCP/IP and UDP/IP transports.)

### 7.4. 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.

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.

### 7.5. Message Authentication

The RPC protocol provides the fields necessary for a client to identify himself to a service and vice versa. Security and access control mechanisms can be built on top of the message authentication.

## 8. RPC Protocol Requirements

The RPC protocol must provide for the following:

1. Unique specification of a procedure to be called.
2. Provisions for matching response messages to request messages.
3. 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:

1. RPC protocol mismatches.
2. Remote program protocol version mismatches.
3. Protocol errors (such as misspecification of a procedure's parameters).
4. Reasons why remote authentication failed.
5. Any other reasons why the desired procedure was not called.

### 8.1. 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). 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).

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

1. The remote implementation of RPC does speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
2. The remote program is not available on the remote system.
3. The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
4. The requested procedure number does not exist (this is usually a caller side protocol or programming error).
5. 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.)

### 8.2. 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
        /* and more to be defined */
};

struct opaque_auth {
        union switch (enum auth_flavor) {
                default: string auth_body<400>;
        };
};
```

In simple English, any `opaque_auth` structure is an `auth_flavor` enumeration followed by a counted string, whose bytes are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. Appendix A defines three authentication protocols.

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

### 8.3. Program Number Assignment

Program numbers are given out in groups of 0x20000000 (536870912) according to the following chart:

```
        0 - 1fffffff    defined by Sun
20000000 - 3fffffff    defined by user
40000000 - 5fffffff    transient
60000000 - 7fffffff    reserved
80000000 - 9fffffff    reserved
a0000000 - bfffffff    reserved
c0000000 - dfffffff    reserved
e0000000 - ffffffff    reserved
```

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

### 8.4. 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, or perhaps 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.

### 8.4.1. Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching uses reliable bytes stream protocols (like TCP/IP) for their 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).

### 8.4.2. 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 their transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors.

### 9. The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top down style. Note: 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 {
        SUCCESS = 0,       /* RPC executed successfully */
        PROG_UNAVAIL = 1, /* remote hasn't exported program */
        PROG_MISMATCH= 2, /* remote can't support version # */
        PROC_UNAVAIL = 3, /* program can't support procedure */
        GARBAGE_ARGS = 4  /* procedure can't decode params */
};

/*
 * Reasons why a call message was rejected:
 */
enum reject_stat {
        RPC_MISMATCH = 0, /* RPC version number != 2 */
        AUTH_ERROR = 1    /* remote can't authenticate caller */
};

/*
 * Why authentication failed:
 */
enum auth_stat {
        AUTH_BADCRED = 1,     /* bad credentials (seal broken) */
        AUTH_REJECTEDCRED=2, /* client must begin new session */
        AUTH_BADVERF = 3,     /* bad verifier (seal broken) */
        AUTH_REJECTEDVERF=4, /* verifier expired or replayed */
        AUTH_TOOWEAK = 5,     /* rejected for security reasons */
};

/*
 * 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; the service side cannot
 * treat this id as any type of sequence number.
 */
struct rpc_msg {
```

```
        unsigned        xid;
        union switch (enum msg_type) {
                CALL:   struct call_body;
                REPLY:  struct reply_body;
        };
};
/*
 * Body of an RPC request call:
 * In version 2 of the RPC protocol specification, rpcvers
 * must be equal to 2.  The fields prog, vers, and proc
 * specify the remote program, its version number, and the
 * procedure within the remote program to be called.  After
 * these fields are two  authentication parameters: cred
 * (authentication credentials) and verf (authentication
 * verifier).  The two  authentication parameters are
 * followed by the parameters to the remote procedure,
 * which are specified by the specific program protocol.
 */
struct call_body {
        unsigned rpcvers;       /* must be equal to two (2) */
        unsigned prog;
        unsigned vers;
        unsigned proc;
        struct opaque_auth cred;
        struct opaque_auth verf;
        /* procedure specific parameters start here */
};
/*
 * Body of a reply to an RPC request.
 * The call message was either accepted or rejected.
 */
struct reply_body {
        union switch (enum reply_stat) {
                MSG_ACCEPTED:   struct accepted_reply;
                MSG_DENIED:     struct rejected_reply;
        };
};
/*
 * Reply to an RPC request that was accepted by the server.
 * Note: there could be an error even though the reques
 * was accepted.  The first field is an authentication
 * verifier that the server generates in order to validate
 * itself to the caller.  It is followed by a union whose
 * discriminant is an enum accept_stat.  The SUCCESS arm
 * of the union is protocol specific.  The PROG_UNAVAIL,
 * PROC_UNAVAIL, and GARBAGE_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 {
        struct opaque_auth      verf;
        union switch (enum accept_stat) {
```

```
                SUCCESS: struct {
                        /*
                         * procedure-specific results start here
                         */
                };
                PROG_MISMATCH: struct {
                        unsigned low;
                        unsigned  high;
                };
                default: struct {
                        /*
                         * void. Cases include PROG_UNAVAIL,
                         * PROC_UNAVAIL, and GARBAGE_ARGS.
                         */
                };
        };
};

/*
 * 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 refuses
 * to authenticate the caller (AUTH_ERROR).  In the case of
 * an RPC version mismatch, the server returns the lowest
 * and highest supported RPC version numbers.  In the case
 * of refused authentication, failure status is returned.
 */
struct rejected_reply {
        union switch (enum reject_stat) {
                RPC_MISMATCH: struct {
                        unsigned low;
                        unsigned high;
                };
                AUTH_ERROR: enum auth_stat;
        };
};
```

### 9.1. Authentication Parameter Specification

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some "flavors" of authentication which have been implemented at (and supported by) Sun.

#### 9.1.1. 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 auth_body string are undefined. It is recommended that the string length be zero.

### 9.1.2. 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 string encode the the following (XDR) structure:

```
struct auth_unix {
        unsigned        stamp;
        string          machinename<255>;
        unsigned        uid;
        unsigned        gid;
        unsigned        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` . In the case of `AUTH_SHORT`, the bytes of the `"response verifier"`'s string encode an `auth_opaque` structure. This new `auth_opaque` structure may now be passed to the server instead of the original `AUTH_UNIX` flavor credentials. The server keeps a cache which maps short hand `auth_opaque` structures (passed back by way of 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 short hand `auth_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.

### 9.2. 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). Sun 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!)

# 1. Appendix A: Port Mapper Program Protocol

The port mapper program maps RPC program and version numbers to UDP/IP or TCP/IP 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.

## 1.1. The RPC Protocol

The protocol is specified by the XDR description language.

```
Port Mapper RPC Program Number: 100000
        Version Number: 1
        Supported Transports:
                UDP/IP on port 111
                RM/TCP/IP on port 111
```

### 1.1.1. Transport Protocol Numbers

```
#define IPPROTO_TCP      6       /* protocol number for TCP/IP */
#define IPPROTO_UDP      17      /* protocol number for UDP/IP */
```

### 1.1.2. RPC Procedures

Here is a list of RPC procedures:

#### 1.1.2.1. Do Nothing

Procedure 0, Version 2.

```
0. PMAPPROC_NULL () returns ()
```

This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results.

#### 1.1.2.2. Set a Mapping

Procedure 1, Version 2.

```
1. PMAPPROC_SET (prog,vers,prot,port) returns (resp)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned port;
        boolean resp;
```

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 `resp`, 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]`.

#### 1.1.2.3. Unset a Mapping

Procedure 2, Version 2.

```
2. PMAPPROC_UNSET (prog,vers,dummy1,dummy2) returns (resp)
        unsigned prog;
        unsigned vers;
        unsigned dummy1;  /* value always ignored */
        unsigned dummy2;  /* value always ignored */
        boolean resp;
```

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`.

#### 1.1.2.4. Look Up a Mapping

Procedure 3, Version 2.

```
3. PMAPPROC_GETPORT (prog,vers,prot,dummy) returns (port)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned dummy; /* this value always ignored */
        unsigned port;  /* zero means program not registered */
```

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 zeros means the program has not been registered.

#### 1.1.2.5. Dumping the Mappings

Procedure 4, Version 2.

```
4. PMAPPROC_DUMP () returns (maplist)
        struct maplist {
                union switch (boolean) {
                        FALSE: struct { /* void, end of list */ };
                        TRUE: struct {
                                unsigned prog;
                                unsigned vers;
                                unsigned prot;
                                unsigned port;
                                struct maplist the_rest;
                        };
                };
        } maplist;
```

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.

#### 1.1.2.6. Indirect Call Routine

Procedure 5, Version 2.

```
5. PMAPPROC_CALLIT (prog,vers,proc,args) returns (port,res)
        unsigned prog;
        unsigned vers;
        unsigned proc;
        string args<>;
        unsigned port;
        string res<>;
```

This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's port number. Its intended use is 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:

1. This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise.
2. 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.

## Appendix D: Mail Systems and Addressing in 4.2bsd

Eric Allman†

*Britton-Lee, Inc.*
*1919 Addison Street, Suite 105.*
*Berkeley, California 94704.*

eric@Berkeley.ARPA
ucbvax!eric

ABSTRACT

Routing mail through a heterogeneous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an ad hoc basis. However, this approach has become unmanageable as internets grow.

Sendmail acts a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both old and new format addresses. The new format is "domain-based," a flexible technique that can handle many common situations. Sendmail is not intended to perform user interface functions.

Sendmail will replace delivermail in the Berkeley 4.2 distribution. Several major hosts are now or will soon be running sendmail. This change will affect any users that route mail through a sendmail gateway. The changes that will be user visible are emphasized.

The mail system to appear in 4.2bsd will contain a number of changes. Most of these changes are based on the replacement of *delivermail* with a new module called *sendmail*. *Sendmail* implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration. Of key interest to the mail system user will be the changes in the network addressing structure.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characteristics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require that the route the message takes be explicitly specified by the sender, simplifying the database update problem since only adjacent hosts must be entered into the system tables, while others use logical addressing, where the sender specifies the location of the recipient but not how to get there. Some networks use a left-associative syntax and others use a right-

†A considerable part of this work was done while under the employ of the INGRES Project at the University of California at Berkeley.

associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, username} triples. Network numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was changed to address domains, comprised of a local resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@a.cc.berkeley.arpa" describes the logical organization of the address space (user "eric" on host "a" in the Computer Center at Berkeley) but not the physical networks used (for example, this could go over different networks depending on whether "a" were on an ethernet or a store-and-forward network).

*Sendmail* is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 defines some of the terms frequently left fuzzy when working in mail systems. Section 2 discusses the design goals for *sendmail*. In section 3, the new address formats and basic features of *sendmail* are described. Section 4 discusses some of the special problems of the UUCP network. The differences between *sendmail* and *delivermail* are presented in section 5.

**DISCLAIMER:** A number of examples in this paper use names of actual people and organizations. This is not intended to imply a commitment or even an intellectual agreement on the part of these people or organizations. In particular, Bell Telephone Laboratories (BTL), Digital Equipment Corporation (DEC), Lawrence Berkeley Laboratories (LBL), Britton-Lee Incorporated (BLI), and the University of California at Berkeley are not committed to any of these proposals at this time. Much of this paper represents no more than the personal opinions of the author.

## 1. DEFINITIONS

There are four basic concepts that must be clearly distinguished when dealing with mail systems: the user (or the user's agent), the user's identification, the user's address, and the route. These are distinguished primarily by their position independence.

### 1.1. User and Identification

The user is the being (a person or program) that is creating or receiving a message. An *agent* is an entity operating on behalf of the user - such as a secretary who handles my mail. or a program that automatically returns a message such as "I am at the UNICOM conference."

The identification is the tag that goes along with the particular user. This tag is completely independent of location. For example, my identification is the string "Eric Allman," and this identification does not change whether I am located at U.C. Berkeley, at Britton-Lee, or at a scientific institute in Austria.

Since the identification is frequently ambiguous (e.g., there are two "Robert Henry"s at Berkeley) it is common to add other disambiguating information that is not strictly part of the identification (e.g., Robert "Code Generator" Henry versus Robert "System Administrator" Henry).

### 1.2. Address

The address specifies a location. As I move around, my address changes. For example, my address might change from "eric@Berkeley.ARPA" to "eric@bli.UUCP" or "allman@IIASA.Austria" depending on my current affiliation.

However, an address is independent of the location of anyone else. That is, my address remains the same to everyone who might be sending me mail. For example, a person at MIT and a person at USC could both send to "eric@Berkeley.ARPA" and have it arrive to the same mailbox.

Ideally a "white pages" service would be provided to map user identifications into addresses (for example, see [Solomon81]). Currently this is handled by passing around scraps of paper or by calling people on the telephone to find out their address.

### 1.3. Route

While an address specifies *where* to find a mailbox, a route specifies *how* to find the mailbox. Specifically, it specifies a path from sender to receiver. As such, the route is potentially different for every pair of people in the electronic universe.

Normally the route is hidden from the user by the software. However, some networks put the burden of determining the route onto the sender. Although this simplifies the software, it also greatly impairs the usability for most users. The UUCP network is an example of such a network.

## 2. DESIGN GOALS

Design goals for *sendmail*[1] include:

(1) Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail, Berkeley *Mail* [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78]. ARPANET mail [Crocker82] was also required.

(2) Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formated addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really *are* used and are difficult to support, but must be supported.

(3) Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.

(4) Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ethernets). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use.

(5) Configuration information should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program

[1] This section makes no distinction between *delivermail* and *sendmail*.

gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.

(6) *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias file.

(7) Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and facilitates specialized functions (such as returning an "I am on vacation" message).

(8) Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

These goals motivated the architecture illustrated in figure 1. The user interacts with a mail generating and sending program. When the mail is created, the generator calls *sendmail*, which routes the message to the correct mailer(s). Since some of the senders may be network servers and some of the mailers may be network clients, *sendmail* may be used as an internet mail gateway.

## 3. USAGE

### 3.1. Address Formats

Arguments may be flags or addresses. Flags set various processing options. Following flag arguments, address arguments may be given. Addresses follow the syntax in RFC822 [Crocker82] for ARPANET address formats. In brief, the format is:

(1) Anything in parentheses is thrown away (as a comment).

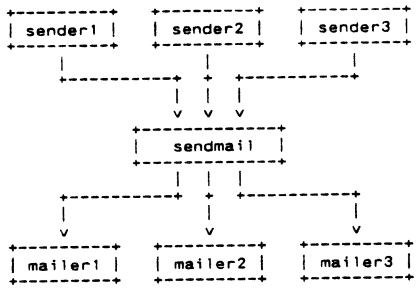



Figure 1 – Sendmail System Structure.

(2) Anything in angle brackets ("< >") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form

user name <machine-address>

will send to the electronic "machine-address" rather than the human "user name."

(3) Double quotes ( " ) quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently - for example, *user* and "*user*" are equivalent, but *user* is different from either of them. This might be used to avoid normal aliasing or duplicate suppression algorithms.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing[2].

Although old style addresses are still accepted in most cases, the preferred address format is based on ARPANET-style domain-based addresses [Su82a]. These addresses are based on a hierarchical, logical decomposition of the address space. The addresses are hierarchical in a sense similar to the U.S. postal addresses: the messages may first be routed to the correct state, with no initial consideration of the city or other addressing details. The addresses are logical in that each step in the hierarchy corresponds to a set of "naming authorities" rather than a physical network.

For example, the address:

eric@HostA.BigSite.ARPA

would first look up the domain BigSite in the namespace administrated by ARPA. A query could then be sent to BigSite for interpretation of HostA. Eventually the mail would arrive at HostA, which would then do final delivery to user "eric."

### 3.2. Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msgs* program).

Any address passing through the initial parsing algorithm as a local address (i.e, not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar ("|") the rest of the address is processed as a shell command. If the user name begins with a slash mark ("/") the name is used as a file name, instead of a login name.

### 3.3. Aliasing, Forwarding, Inclusion

*Sendmail* reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

#### 3.3.1. Aliasing

Aliasing maps local addresses to address lists using a system-wide file. This file is hashed to speed access. Only addresses that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

[2] Disclaimer: Some special processing is done after rewriting local names, see below.

### 3.3.2. Forwarding

After aliasing, if an recipient address specifies a local user *sendmail* searches for a ".forward" file in the recipient's home directory. If it exists, the message is *not* sent to that user, but rather to the list of addresses in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

"|/usr/local/newmail myname"

will use a different incoming mailer.

### 3.3.3. Inclusion

Inclusion is specified in RFC 733 [Crocker77] syntax:

:Include: pathname

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

project: :include:/usr/project/userlist

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

It is not necessary to rebuild the index on the alias database when a :include: list is changed.

## 3.4. Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line. The body is an uninterpreted sequence of text lines.

The header is formated as a series of lines of the form

field-name: field-value

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

# 4. THE UUCP PROBLEM

Of particular interest is the UUCP network. The explicit routing used in the UUCP environment causes a number of serious problems. First, giving out an address is impossible without knowing the address of your potential correspondent. This is typically handled by specifying the address relative to some "well-known" host (e.g., ucbvax or decvax). Second, it is often difficult to compute the set of addresses to reply to without some knowledge of the topology of the network. Although it may be easy for a human being to do this under many circumstances, a program does not have equally sophisticated heuristics built in. Third, certain addresses will become painfully and unnecessarily long, as when a message is routed through many hosts in the USENET. And finally, certain "mixed domain" addresses are impossible to parse unambiguously – e.g.,

decvax!ucbvax!lbl-h!user@LBL-CSAM

might have many possible resolutions, depending on whether the message was first routed to decvax or to LBL-CSAM.

To solve this problem, the UUCP syntax would have to be changed to use addresses rather than routes. For example, the address "decvax!ucbvax!eric" might be expressed as "eric@ucbvax.UUCP" (with the hop through decvax implied). This address would itself be a domain-based address; for example, an address might be of the form:

mark@d.cbosg.btl.UUCP

Hosts outside of Bell Telephone Laboratories would then only need to know how to get to a designated BTL relay, and the BTL topology would only be maintained inside Bell.

There are three major problems associated with turning UUCP addresses into something reasonable: defining the namespace, creating and propagating the necessary software, and building and maintaining the database.

### 4.1. Defining the Namespace

Putting all UUCP hosts into a flat namespace (e.g., "...@host.UUCP") is not practical for a number of reasons. First, with over 1600 sites already, and (with the increasing availability of inexpensive microcomputers and autodialers) several thousand more coming within a few years, the database update problem is simply intractable if the namespace is flat. Second, there are almost certainly name conflicts today. Third, as the number of sites grow the names become ever less mnemonic.

It seems inevitable that there be some sort of naming authority for the set of top level names in the UUCP domain, as unpleasant a possibility as that may seem. It will simply not be possible to have one host resolving all names. It may however be possible to handle this in a fashion similar to that of assigning names of newsgroups in USENET. However, it will be essential to encourage everyone to become subdomains of an existing domain whenever possible - even though this will certainly bruise some egos. For example, if a new host named "blid" were to be added to the UUCP network, it would probably actually be addressed as "d.bli.UUCP" (i.e., as host "d" in the pseudo-domain "bli" rather than as host "blid" in the UUCP domain).

### 4.2. Creating and Propagating the Software

The software required to implement a consistent namespace is relatively trivial. Two modules are needed, one to handle incoming mail and one to handle outgoing mail.

The incoming module must be prepared to handle either old or new style addresses. New-style addresses can be passed through unchanged. Old style addresses must be turned into new style addresses where possible.

The outgoing module is slightly trickier. It must do a database lookup on the recipient addresses (passed on the command line) to determine what hosts to send the message to. If those hosts do not accept new-style addresses, it must transform all addresses in the header of the message into old style using the database lookup.

Both of these modules are straightforward except for the issue of modifying the header. It seems prudent to choose one format for the message headers. For a number of reasons, Berkeley has elected to use the ARPANET protocols for message formats. However, this protocol is somewhat difficult to parse.

Propagation is somewhat more difficult. There are a large number of hosts connected to UUCP that will want to run completely standard systems (for very good reasons). The strategy is not to convert the entire network - only enough of it it alleviate the problem.

### 4.3. Building and Maintaining the Database

This is by far the most difficult problem. A prototype for this database already exists, but it is maintained by hand and does not pretend to be complete.

This problem will be reduced considerably if people choose to group their hosts into subdomains. This would require a global update only when a new top level domain joined the network. A message to a host in a subdomain could simply be routed to a known domain gateway for further processing. For example, the address "eric@a.bli.UUCP" might be routed to the "bli" gateway for redistribution; new hosts could be added within BLI without notifying the rest of the world. Of course, other hosts *could* be notified as an efficiency measure.

There may be more than one domain gateway. A domain such as BTL, for instance, might have a dozen gateways to the outside world; a non-BTL site could choose the closest gateway. The only restriction would be that all gateways maintain a consistent view of the domain they represent.

### 4.4. Logical Structure

Logically, domains are organized into a tree. There need not be a host actually associated with each level in the tree - for example, there will be no host associated with the name "UUCP." Similarly, an organization might group names together for administrative reasons; for example, the name

CAD.research.BigCorp.UUCP

might not actually have a host representing "research."

However, it may frequently be convenient to have a host or hosts that "represent" a domain. For example, if a single host exists that represents Berkeley, then mail from outside Berkeley can forward mail to that host for further resolution without knowing Berkeley's (rather volatile) topology. This is not unlike the operation of the telephone network.

This may also be useful inside certain large domains. For example, at Berkeley it may be presumed that most hosts know about other hosts inside the Berkeley domain. But if they process an address that is unknown, they can pass it "upstairs" for further examination. Thus as new hosts are added only one host (the domain master) *must* be updated immediately; other hosts can be updated as convenient.

Ideally this name resolution process would be performed by a name server (e.g., [Su82b]) to avoid unnecessary copying of the message. However, in a batch network such as UUCP this could result in unnecessary delays.

## 5. COMPARISON WITH DELIVERMAIL

*Sendmail* is an outgrowth of *delivermail*. The primary differences are:

(1) Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.

(2) Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.

(3) Forwarding and :include: features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).

(4) *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.

(5) A mail queue is provided in *sendmail*. Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it may be reliably redelivered even if the system crashes during the initial delivery.

(6) *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

## REFERENCES


[Crocker77] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages*. RFC 733, NIC 41952. In [Feinler78]. November 1977.

[Crocker82] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages*. RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Feinler78] Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook*. NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.

[Nowitz78] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems*. Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[Schmidt79] Schmidt, E., *An Introduction to the Berkeley Network*. University of California, Berkeley California. 1979.

[Shoens79] Shoens, K., *Mail Reference Manual*. University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.

[Solomon81] Solomon, M., Landweber, L., and Neuhengen, D., *The Design of the CSNET Name Server*. CS-DN-2. University of Wisconsin, Madison. October 1981.

[Su82a] Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications*. RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Su82b] Su, Zaw-Sing, *A Distributed System for Internet Name Service*. RFC830. Network Information Center, SRI International, Menlo Park, California. October 1982.

# Appendix E: SENDMAIL – An Internetwork Mail Router

Eric Allman†

*Britton-Lee, Inc.*
*1919 Addison Street, Suite 105*
*Berkeley, California 94704*

ABSTRACT

Routing mail through a heterogenous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an *ad hoc* basis. However, this approach has become unmanageable as internets grow.

Sendmail acts a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both domain-based addressing and old-style *ad hoc* addresses. The production system is powerful enough to rewrite addresses in the message header to conform to the standards of a number of common target networks, including old (NCP/RFC733) Arpanet, new (TCP/RFC822) Arpanet, UUCP, and Phonenet. Sendmail also implements an SMTP server, message queueing, and aliasing.

*Sendmail* implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characterstics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require point-to-point routing, which simplifies the database update problem since only adjacent hosts must be entered into the system tables, while others use end-to-end addressing. Some networks use a left-associative syntax and others use a right-associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, resource} triples. Network numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was quickly expanded to address domains, comprised of a local resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@a.cc.berkeley.arpa" describes only the logical

†A considerable part of this work was done while under the employ of the INGRES Project at the University of California at Berkeley.

organization of the address space.

*Sendmail* is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 discusses the design goals for *sendmail*. Section 2 gives an overview of the basic functions of the system. In section 3, details of usage are discussed. Section 4 compares *sendmail* to other internet mail routers, and an evaluation of *sendmail* is given in section 5, including future plans.

## 1. DESIGN GOALS

Design goals for *sendmail* include:

(1) Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail [UNIX83], Berkeley *Mail* [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78a, Nowitz78b]. ARPANET mail [Crocker77a, Postel77] was also required.

(2) Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formated addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with UNIX mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really *are* used and are difficult to support, but must be supported.

(3) Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.

(4) Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ether nets [Metcalfe76]). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use. For example, the ARPANET is bringing up the TCP protocol to replace the old NCP protocol. No host at Berkeley runs both TCP and NCP, so it is necessary to look at the ARPANET host name to determine whether to route mail to an NCP gateway or a TCP gateway.

(5) Configuration should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.

(6) *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias file.

(7) Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and

facilitates specialized functions (such as returning an "I am on vacation" message).

(8) Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

These goals motivated the architecture illustrated in figure 1. The user interacts with a mail generating and sending program. When the mail is created, the generator calls *sendmail*, which routes the message to the correct mailer(s). Since some of the senders may be network servers and some of the mailers may be network clients, *sendmail* may be used as an internet mail gateway.

## 2. OVERVIEW

### 2.1. System Organization

*Sendmail* neither interfaces with the user nor does actual mail delivery. Rather, it collects a message generated by a user interface program (UIP) such as Berkeley *Mail*, MS [Crocker77b], or MH [Borden79], edits the message as required by the destination network, and calls appropriate mailers to do mail delivery or queueing for network transmission[1]. This discipline allows the insertion of new mailers at minimum cost. In this sense *sendmail* resembles the Message Processing Module (MPM) of [Postel79b].

### 2.2. Interfaces to the Outside World

There are three ways *sendmail* can communicate with the outside world, both in receiving and in sending mail. These are using the conventional UNIX argument vector/return status, speaking SMTP over a pair of UNIX pipes, and speaking SMTP

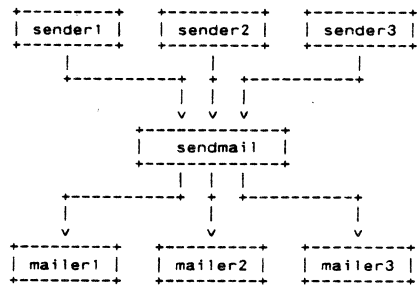



Figure 1 – Sendmail System Structure.

[1] except when mailing to a file, when *sendmail* does the delivery directly.

over an interprocess(or) channel.

#### 2.2.1. Argument vector/exit status

This technique is the standard UNIX method for communicating with the process. A list of recipients is sent in the argument vector, and the message body is sent on the standard input. Anything that the mailer prints is simply collected and sent back to the sender if there were any problems. The exit status from the mailer is collected after the message is sent, and a diagnostic is printed if appropriate.

#### 2.2.2. SMTP over pipes

The SMTP protocol [Postel82] can be used to run an interactive lock-step interface with the mailer. A subprocess is still created, but no recipient addresses are passed to the mailer via the argument list. Instead, they are passed one at a time in commands sent to the processes standard input. Anything appearing on the standard output must be a reply code in a special format.

#### 2.2.3. SMTP over an IPC connection

This technique is similar to the previous technique, except that it uses a 4.2bsd IPC channel [UNIX83]. This method is exceptionally flexible in that the mailer need not reside on the same machine. It is normally used to connect to a sendmail process on another machine.

### 2.3. Operational Description

When a sender wants to send a message, it issues a request to *sendmail* using one of the three methods described above. *Sendmail* operates in two distinct phases. In the first phase, it collects and stores the message. In the second phase, message delivery occurs. If there were errors during processing during the second phase, *sendmail* creates and returns a new message describing the error and/or returns an status code telling what went wrong.

#### 2.3.1. Argument processing and address parsing

If *sendmail* is called using one of the two subprocess techniques, the arguments are first scanned and option specifications are processed. Recipient addresses are then collected, either from the command line or from the SMTP RCPT command, and a list of recipients is created. Aliases are expanded at this step, including mailing lists. As much validation as possible of the addresses is done at this step: syntax is checked, and local addresses are verified, but detailed checking of host names and addresses is deferred until delivery. Forwarding is also performed as the local addresses are verified.

*Sendmail* appends each address to the recipient list after parsing. When a name is aliased or forwarded, the old name is retained in the list, and a flag is set that tells the delivery phase to ignore this recipient. This list is kept free from duplicates, preventing alias loops and duplicate messages deliverd to the same recipient, as might occur if a person is in two groups.

#### 2.3.2. Message collection

*Sendmail* then collects the message. The message should have a header at the beginning. No formatting requirements are imposed on the message except that they must be lines of text (i.e., binary data is not allowed). The header is parsed and stored in memory, and the body of the message is saved in a temporary file.

To simplify the program interface, the message is collected even if no addresses were valid. The message will be returned with an error.

#### 2.3.3. Message delivery

For each unique mailer and host in the recipient list, *sendmail* calls the appropriate mailer. Each mailer invocation sends to all users receiving the message on one host. Mailers that only accept one recipient at a time are handled properly.

The message is sent to the mailer using one of the same three interfaces used to submit a message to sendmail. Each copy of the message is prepended by a customized header. The mailer status code is caught and checked, and a suitable error message given as appropriate. The exit code must conform to a system standard or a generic message ("Service unavailable") is given.

#### 2.3.4. Queueing for retransmission

If the mailer returned an status that indicated that it might be able to handle the mail later, *sendmail* will queue the mail and try again later.

#### 2.3.5. Return to sender

If errors occur during processing, *sendmail* returns the message to the sender for retransmission. The letter can be mailed back or written in the file "dead.letter" in the sender's home directory[2].

### 2.4. Message Header Editing

Certain editing of the message header occurs automatically. Header lines can be inserted under control of the configuration file. Some lines can be merged; for example, a "From:" line and a "Full-name:" line can be merged under certain circumstances.

### 2.5. Configuration File

Almost all configuration information is read at runtime from an ASCII file, encoding macro definitions (defining the value of macros used internally), header declarations (telling sendmail the format of header lines that it will process specially, i.e., lines that it will add or reformat), mailer definitions (giving information such as the location and characteristics of each mailer), and address rewriting rules (a limited production system to rewrite addresses which is used to parse and rewrite the addresses).

To improve performance when reading the configuration file, a memory image can be provided. This provides a "compiled" form of the configuration file.

## 3. USAGE AND IMPLEMENTATION

### 3.1. Arguments

Arguments may be flags and addresses. Flags set various processing options. Following flag arguments, address arguments may be given, unless we are running in SMTP mode. Addresses follow the syntax in RFC822 [Crocker82] for ARPANET address formats. In brief, the format is:

---

[2] Obviously, if the site giving the error is not the originating site, the only reasonable option is to mail back to the sender. Also, there are many more error disposition options, but they only effect the error message – the "return to sender" function is always handled in one of these two ways.

(1) Anything in parentheses is thrown away (as a comment).

(2) Anything in angle brackets ("< >") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form

user name <machine-address>

will send to the electronic "machine-address" rather than the human "user name."

(3) Double quotes ( " ) quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently – for example, *user* and "*user*" are equivalent, but *user* is different from either of them.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing[3].

### 3.2. Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msgs* program, or the MARS system [Sattley78]).

Any address passing through the initial parsing algorithm as a local address (i.e, not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar ("|") the rest of the address is processed as a shell command. If the user name begins with a slash mark ("/") the name is used as a file name, instead of a login name.

Files that have setuid or setgid bits set but no execute bits set have those bits honored if *sendmail* is running as root.

### 3.3. Aliasing, Forwarding, Inclusion

*Sendmail* reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

#### 3.3.1. Aliasing

Aliasing maps names to address lists using a system-wide file. This file is indexed to speed access. Only names that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

#### 3.3.2. Forwarding

After aliasing, recipients that are local and valid are checked for the existence of a ".forward" file in their home directory. If it exists, the message is *not* sent to that user, but rather to the list of users in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

"|/usr/local/newmail myname"

---

[3] Disclaimer: Some special processing is done after rewriting local names; see below.

will use a different incoming mailer.

### 3.3.3. Inclusion

Inclusion is specified in RFC 733 [Crocker77a] syntax:

:Include: pathname

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

project: :include:/usr/project/userlist

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

It is not necessary to rebuild the index on the alias database when a :include: list is changed.

## 3.4. Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line.

The header is formatted as a series of lines of the form

field-name: field-value

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

The body is a series of text lines. It is completely uninterpreted and untouched, except that lines beginning with a dot have the dot doubled when transmitted over an SMTP channel. This extra dot is stripped by the receiver.

## 3.5. Message Delivery

The send queue is ordered by receiving host before transmission to implement message batching. Each address is marked as it is sent so rescanning the list is safe. An argument list is built as the scan proceeds. Mail to files is detected during the scan of the send list. The interface to the mailer is performed using one of the techniques described in section 2.2.

After a connection is established, *sendmail* makes the per-mailer changes to the header and sends the result to the mailer. If any mail is rejected by the mailer, a flag is set to invoke the return-to-sender function after all delivery completes.

## 3.6. Queued Messages

If the mailer returns a "temporary failure" exit status, the message is queued. A control file is used to describe the recipients to be sent to and various other parameters. This control file is formatted as a series of lines, each describing a sender, a recipient, the time of submission, or some other salient parameter of the message. The header of the message is stored in the control file, so that the associated data file in the queue is just the temporary file that was originally collected.

### 3.7. Configuration

Configuration is controlled primarily by a configuration file read at startup. *Sendmail* should not need to be recomplied except

(1) To change operating systems (V6, V7/32V, 4BSD).

(2) To remove or insert the DBM (UNIX database) library.

(3) To change ARPANET reply codes.

(4) To add headers fields requiring special processing.

Adding mailers or changing parsing (i.e., rewriting) or routing information does not require recompilation.

If the mail is being sent by a local user, and the file ".mailcf" exists in the sender's home directory, that file is read as a configuration file after the system configuration file. The primary use of this feature is to add header lines.

The configuration file encodes macro definitions, header definitions, mailer definitions, rewriting rules, and options.

#### 3.7.1. Macros

Macros can be used in three ways. Certain macros transmit unstructured textual information into the mail system, such as the name *sendmail* will use to identify itself in error messages. Other macros transmit information from *sendmail* to the configuration file for use in creating other fields (such as argument vectors to mailers); e.g., the name of the sender, and the host and user of the recipient. Other macros are unused internally, and can be used as shorthand in the configuration file.

#### 3.7.2. Header declarations

Header declarations inform *sendmail* of the format of known header lines. Knowledge of a few header lines is built into *sendmail*, such as the "From:" and "Date:" lines.

Most configured headers will be automatically inserted in the outgoing message it they don't exist in the incoming message. Certain headers are suppressed by some mailers.

#### 3.7.3. Mailer declarations

Mailer declarations tell *sendmail* of the various mailers available to it. The definition specifies the internal name of the mailer, the pathname of the program to call, some flags associated with the mailer, and an argument vector to be used on the call; this vector is macro-expanded before use.

#### 3.7.4. Address rewriting rules

The heart of address parsing in *sendmail* is a set of rewriting rules. These are an ordered list of pattern-replacement rules, (somewhat like a production system, except that order is critical), which are applied to each address. The address is rewritten textually until it is either rewritten into a special canonical form (i.e., a (mailer, host, user) 3-tuple, such as {arpanet, usc-isif, postel} representing the address "postel@usc-isif"), or it falls off the end. When a pattern matches, the rule is reapplied until it fails.

The configuration file also supports the editing of addresses into different formats. For example, an address of the form:

ucsfcgl!tef

might be mapped into:

tef@ucsfcgl.UUCP

to conform to the domain syntax. Translations can also be done in the other direction.

### 3.7.5. Option setting

There are several options that can be set from the configuration file. These include the pathnames of various support files, timeouts, default modes, etc.

## 4. COMPARISON WITH OTHER MAILERS

### 4.1. Delivermail

*Sendmail* is an outgrowth of *delivermail*. The primary differences are:

(1) Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.

(2) Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.

(3) Forwarding and :include: features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).

(4) *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.

(5) A mail queue is provided in *sendmail*. Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it may be reliably redelivered even if the system crashes during the initial delivery.

(6) *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

### 4.2. MMDF

MMDF [Crocker79] spans a wider problem set than *sendmail*. For example, the domain of MMDF includes a "phone network" mailer, whereas *sendmail* calls on preexisting mailers in most cases.

MMDF and *sendmail* both support aliasing, customized mailers, message batching, automatic forwarding to gateways, queueing, and retransmission. MMDF supports two-stage timeout, which *sendmail* does not support.

The configuration for MMDF is compiled into the code[1].

Since MMDF does not consider backwards compatibility as a design goal, the address parsing is simpler but much less flexible.

---

[1] Dynamic configuration tables are currently being considered for MMDF; allowing the installer to select either compiled or dynamic tables.

It is somewhat harder to integrate a new channel[5] into MMDF. In particular, MMDF must know the location and format of host tables for all channels, and the channel must speak a special protocol. This allows MMDF to do additional verification (such as verifying host names) at submission time.

MMDF strictly separates the submission and delivery phases. Although *sendmail* has the concept of each of these stages, they are integrated into one program, whereas in MMDF they are split into two programs.

### 4.3. Message Processing Module

The Message Processing Module (MPM) discussed by Postel [Postel79b] matches *sendmail* closely in terms of its basic architecture. However, like MMDF, the MPM includes the network interface software as part of its domain.

MPM also postulates a duplex channel to the receiver, as does MMDF, thus allowing simpler handling of errors by the mailer than is possible in *sendmail*. When a message queued by *sendmail* is sent, any errors must be returned to the sender by the mailer itself. Both MPM and MMDF mailers can return an immediate error response, and a single error processor can create an appropriate response.

MPM prefers passing the message as a structured object, with type-length-value tuples[6]. Such a convention requires a much higher degree of cooperation between mailers than is required by *sendmail*. MPM also assumes a universally agreed upon internet name space (with each address in the form of a net-host-user tuple), which *sendmail* does not.

## 5. EVALUATIONS AND FUTURE PLANS

*Sendmail* is designed to work in a nonhomogeneous environment. Every attempt is made to avoid imposing unnecessary constraints on the underlying mailers. This goal has driven much of the design. One of the major problems has been the lack of a uniform address space, as postulated in [Postel79a] and [Postel79b].

A nonuniform address space implies that a path will be specified in all addresses, either explicitly (as part of the address) or implicitly (as with implied forwarding to gateways). This restriction has the unpleasant effect of making replying to messages exceedingly difficult, since there is no one "address" for any person, but only a way to get there from wherever you are.

Interfacing to mail programs that were not initially intended to be applied in an internet environment has been amazingly successful, and has reduced the job to a manageable task.

*Sendmail* has knowledge of a few difficult environments built in. It generates ARPANET FTP/SMTP compatible error messages (prepended with three-digit numbers [Neigus73, Postel74, Postel82]) as necessary, optionally generates UNIX-style "From" lines on the front of messages for some mailers, and knows how to parse the same lines on input. Also, error handling has an option customized for BerkNet.

The decision to avoid doing any type of delivery where possible (even, or perhaps especially, local delivery) has turned out to be a good idea. Even with local delivery, there are issues of the location of the mailbox, the format of the mailbox, the locking protocol used, etc., that are best decided by other programs. One surprisingly major annoyance in many internet mailers is that the location and format of local mail is built in. The feeling seems to be that local mail is so common that it should be efficient. This feeling is not

[5] The MMDF equivalent of a *sendmail* "mailer."

[6] This is similar to the NBS standard.

born out by our experience; on the contrary, the location and format of mailboxes seems to vary widely from system to system.

The ability to automatically generate a response to incoming mail (by forwarding mail to a program) seems useful ("I am on vacation until late August....") but can create problems such as forwarding loops (two people on vacation whose programs send notes back and forth, for instance) if these programs are not well written. A program could be written to do standard tasks correctly, but this would solve the general case.

It might be desirable to implement some form of load limiting. I am unaware of any mail system that addresses this problem, nor am I aware of any reasonable solution at this time.

The configuration file is currently practically inscrutable; considerable convenience could be realized with a higher-level format.

It seems clear that common protocols will be changing soon to accommodate changing requirements and environments. These changes will include modifications to the message header (e.g., [NBS80]) or to the body of the message itself (such as for multimedia messages [Postel80]). Experience indicates that these changes should be relatively trivial to integrate into the existing system.

In tightly coupled environments, it would be nice to have a name server such as Grapvine [Birrell82] integrated into the mail system. This would allow a site such as "Berkeley" to appear as a single host, rather than as a collection of hosts, and would allow people to move transparently among machines without having to change their addresses. Such a facility would require an automatically updated database and some method of resolving conflicts. Ideally this would be effective even without all hosts being under a single management. However, it is not clear whether this feature should be integrated into the aliasing facility or should be considered a "value added" feature outside *sendmail* itself.

As a more interesting case, the CSNET name server [Solomon81] provides an facility that goes beyond a single tightly-coupled environment. Such a facility would normally exist outside of *sendmail* however.

## ACKNOWLEDGEMENTS

Thanks are due to Kurt Shoens for his continual cheerful assistance and good advice, Bill Joy for pointing me in the correct direction (over and over), and Mark Horton for more advice, prodding, and many of the good ideas. Kurt and Eric Schmidt are to be credited for using *delivermail* as a server for their programs (*Mail* and BerkNet respectively) before any sane person should have, and making the necessary modifications promptly and happily. Eric gave me considerable advice about the perils of network software which saved me an unknown amount of work and grief. Mark did the original implementation of the DBM version of aliasing, installed the VFORK code, wrote the current version of *rmail*, and was the person who really convinced me to put the work into *delivermail* to turn it into *sendmail*. Kurt deserves accolades for using *sendmail* when I was myself afraid to take the risk; how a person can continue to be so enthusiastic in the face of so much bitter reality is beyond me.

Kurt, Mark, Kirk McKusick, Marvin Solomon, and many others have reviewed this paper, giving considerable useful advice.

Special thanks are reserved for Mike Stonebraker at Berkeley and Bob Epstein at Britton-Lee, who both knowingly allowed me to put so much work into this project when there were so many other things I really should have been working on.

## REFERENCES


[Birrell82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M. 25*, 4, April 82.

[Borden79] Borden, S., Gaines, R. S., and Shapiro, N. Z., *The MH Message Handling System: Users' Manual*. R-2367-PAF. Rand Corporation. October 1979.

[Crocker77a] Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages*. RFC 733, NIC 41952. In [Feinler78]. November 1977.

[Crocker77b] Crocker, D. H., *Framework and Functions of the MS Personal Message System*. R-2134-ARPA, Rand Corporation, Santa Monica, California. 1977.

[Crocker79] Crocker, D. H., Szurkowski, E. S., and Farber, D. J., *An Internetwork Memo Distribution Facility – MMDF*. 6th Data Communication Symposium, Asilomar. November 1979.

[Crocker82] Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages*. RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Metcalfe76] Metcalfe, R., and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM 19, 7*. July 1976.

[Feinler78] Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook*. NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.

[NBS80] National Bureau of Standards, *Specification of a Draft Message Format Standard*. Report No. ICST/CBOS 80-2. October 1980.

[Neigus73] Neigus, N., *File Transfer Protocol for the ARPA Network*. RFC 542, NIC 17759. In [Feinler78]. August, 1973.

[Nowitz78a] Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems*. Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[Nowitz78b] Nowitz, D. A., *Uucp Implementation Description*. Bell Laboratories. In UNIX Programmer's Manual. Seventh Edition, Volume 2. October, 1978.

[Postel74] Postel, J., and Neigus, N., Revised FTP Reply Codes. RFC 640, NIC 30843. In [Feinler78]. June, 1974.

[Postel77] Postel, J., *Mail Protocol*. NIC 29588. In [Feinler78]. November 1977.

[Postel79a] Postel, J., *Internet Message Protocol*. RFC 753, IEN 85. Network Information Center, SRI International, Menlo Park, California. March 1979.

[Postel79b] Postel, J. B., *An Internetwork Message Structure*. In *Proceedings of the Sixth Data Communications Symposium*, IEEE. New York. November 1979.

[Postel80] Postel, J. B., *A Structured Format for Transmission of Multi-Media Documents*. RFC 767. Network Information Center, SRI International,

Menlo Park, California. August 1980.

[Postel82] Postel, J. B., *Simple Mail Transfer Protocol.* RFC821 (obsoleting RFC788). Network Information Center, SRI International, Menlo Park, California. August 1982.

[Schmidt79] Schmidt, E., *An Introduction to the Berkeley Network.* University of California, Berkeley California. 1979.

[Shoens79] Shoens, K., *Mail Reference Manual.* University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.

[Sluizer81] Sluizer, S., and Postel, J. B., *Mail Transfer Protocol.* RFC 780. Network Information Center, SRI International, Menlo Park, California. May 1981.

[Solomon81] Solomon, M., Landweber, L., and Neuhengen, D., "The Design of the CSNET Name Server." CS-DN-2, University of Wisconsin, Madison. November 1981.

[Su82] Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications.* RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.

[UNIX83] *The UNIX Programmer's Manual, Seventh Edition,* Virtual VAX-11 Version, Volume 1. Bell Laboratories, modified by the University of California, Berkeley, California. March, 1983.

# Appendix F: Sendmail Installation and Operation Guide

Eric Allman
Britton-Lee, Inc.

Version 5.1

*Sendmail* implements a general purpose internetwork mail routing facility under the UNIX* operating system. It is not tied to any one transport protocol – its function may be likened to a crossbar switch, relaying messages from one domain into another. In the process, it can do a limited amount of message header editing to put the message into a format that is appropriate for the receiving domain. All of this is done under the control of a configuration file.

Due to the requirements of flexibility for *sendmail*, the configuration file can seem somewhat unapproachable. However, there are only a few basic configurations for most sites, for which standard configuration files have been supplied. Most other configurations can be built by adjusting an existing configuration files incrementally.

Although *sendmail* is intended to run without the need for monitoring, it has a number of features that may be used to monitor or adjust the operation under unusual circumstances. These features are described.

Section one describes how to do a basic *sendmail* installation. Section two explains the day-to-day information you should know to maintain your mail system. If you have a relatively normal site, these two sections should contain sufficient information for you to install *sendmail* and keep it happy. Section three describes some parameters that may be safely tweaked. Section four has information regarding the command line arguments. Section five contains the nitty-gritty information about the configuration file. This section is for masochists and people who must write their own configuration file. The appendixes give a brief but detailed explanation of a number of features not described in the rest of the paper.

The references in this paper are actually found in the companion paper *Sendmail – An Internetwork Mail Router*. This other paper should be read before this manual to gain a basic understanding of how the pieces fit together.

## 1. BASIC INSTALLATION

There are two basic steps to installing sendmail. The hard part is to build the configuration table. This is a file that sendmail reads when it starts up that describes the mailers it knows about, how to parse addresses, how to rewrite the message header, and the settings of various options. Although the configuration table is quite complex, a configuration can usually be built by adjusting an existing off-the-shelf configuration. The second part is actually doing the installation, i.e., creating the necessary files, etc.

The remainder of this section will describe the installation of sendmail assuming you can use one of the existing configurations and that the standard installation parameters are acceptable. All pathnames and examples are given from the root of the *sendmail* subtree.

*UNIX is a trademark of Bell Laboratories.

### 1.1. Off-The-Shelf Configurations

The configuration files are all in the subdirectory *cf* of the sendmail directory. The ones used at Berkeley are in *m4*(1) format; files with names ending ".m4" are *m4* include files, while files with names ending ".mc" are the master files. Files with names ending ".cf" are the *m4* processed versions of the corresponding ".mc" file.

Two off the shelf configuration files are supplied to handle the basic cases: *cf/arpaproto.cf* for Arpanet (TCP) sites and *cf/uucpproto.cf* for UUCP sites. These are *not* in *m4* format. The file you need should be copied to a file with the same name as your system, e.g.,

```
cp uucpproto.cf ucstcgl.cf
```

This file is now ready for installation as */usr/lib/sendmail.cf*.

### 1.2. Installation Using the Makefile

A makefile exists in the root of the *sendmail* directory that will do all of these steps for a 4.2bsd system. It may have to be slightly tailored for use on other systems.

Before using this makefile, you should already have created your configuration file and left it in the file "cf/system.cf" where *system* is the name of your system (i.e., what is returned by *hostname*(1)). If you do not have *hostname* you can use the declaration "HOST=system" *on the make(1) command line. You should also examine the file md/config.m4 and change the m4 macros there to reflect any libraries and compilation flags you may need.*

The basic installation procedure is to type:

```
make
make install
```

in the root directory of the *sendmail* distribution. This will make all binaries and install them in the standard places. The second *make* command must be executed as the superuser (root).

### 1.3. Installation by Hand

Along with building a configuration file, you will have to install the *sendmail* startup into your UNIX system. If you are doing this installation in conjunction with a regular Berkeley UNIX install, these steps will already be complete. Many of these steps will have to be executed as the superuser (root).

#### 1.3.1. lib/libsys.a

The library in lib/libsys.a contains some routines that should in some sense be part of the system library. These are the system logging routines and the new directory access routines (if required). If you are not running the new 4.2bsd directory code and do not have the compatibility routines installed in your system library, you should execute the commands:

```
cd lib
make ndir
```

This will compile and install the 4.2 compatibility routines in the library. You should then type:

```
cd lib    # if required
make
```

This will recompile and fill the library.

### 1.3.2. /usr/lib/sendmail

The binary for sendmail is located in /usr/lib. There is a version available in the source directory that is probably inadequate for your system. You should plan on recompiling and installing the entire system:

```
cd src
rm -f *.o
make
cp sendmail /usr/lib
```

### 1.3.3. /usr/lib/sendmail.cf

The configuration file that you created earlier should be installed in /usr/lib/sendmail.cf:

```
cp cf/system.cf /usr/lib/sendmail.cf
```

### 1.3.4. /usr/ucb/newaliases

If you are running delivermail, it is critical that the *newaliases* command be replaced. This can just be a link to *sendmail*:

```
rm -f /usr/ucb/newaliases
ln /usr/lib/sendmail /usr/ucb/newaliases
```

### 1.3.5. /usr/spool/mqueue

The directory */usr/spool/mqueue* should be created to hold the mail queue. This directory should be mode 777 unless *sendmail* is run setuid, when *mqueue* should be owned by the sendmail owner and mode 755.

### 1.3.6. /usr/lib/aliases*

The system aliases are held in three files. The file "/usr/lib/aliases" is the master copy. A sample is given in "lib/aliases" which includes some aliases which *must* be defined:

```
cp lib/aliases /usr/lib/aliases
```

You should extend this file with any aliases that are apropos to your system.

Normally *sendmail* looks at a version of these files maintained by the *dbm*(3) routines. These are stored in "/usr/lib/aliases.dir" and "/usr/lib/aliases.pag." These can initially be created as empty files, but they will have to be initialized promptly. These should be mode 666 if you are running a reasonably relaxed system:

```
cp /dev/null /usr/lib/aliases.dir
cp /dev/null /usr/lib/aliases.pag
chmod 666 /usr/lib/aliases.*
newaliases
```

### 1.3.7. /usr/lib/sendmail.fc

If you intend to install the frozen version of the configuration file (for quick startup) you should create the file /usr/lib/sendmail.fc and initialize it. This step may be safely skipped.

```
cp /dev/null /usr/lib/sendmail.fc
/usr/lib/sendmail -bz
```

### 1.3.8. /etc/rc

It will be necessary to start up the sendmail daemon when your system reboots. This daemon performs two functions: it listens on the SMTP socket for connections (to receive mail from a remote system) and it processes the queue periodically to insure that mail gets delivered when hosts come up.

Add the following lines to "/etc/rc" (or "/etc/rc.local" as appropriate) in the area where it is starting up the daemons:

```
if [ -f /usr/lib/sendmail ]; then
        (cd /usr/spool/mqueue; rm -f [lnx]f*)
        /usr/lib/sendmail -bd -q30m &
        echo -n ' sendmail' >/dev/console
fi
```

The "cd" and "rm" commands insure that all lock files have been removed; extraneous lock files may be left around if the system goes down in the middle of processing a message. The line that actually invokes *sendmail* has two flags: "-bd" causes it to listen on the SMTP port, and "-q30m" causes it to run the queue every half hour.

If you are not running a version of UNIX that supports Berkeley TCP/IP, do not include the **-bd** flag.

### 1.3.9. /usr/lib/sendmail.hf

This is the help file used by the SMTP **HELP** command. It should be copied from "lib/sendmail.hf":

```
cp lib/sendmail.hf /usr/lib
```

### 1.3.10. /usr/lib/sendmail.st

If you wish to collect statistics about your mail traffic, you should create the file "/usr/lib/sendmail.st":

```
cp /dev/null /usr/lib/sendmail.st
chmod 666 /usr/lib/sendmail.st
```

This file does not grow. It is printed with the program "aux/mailstats."

### 1.3.11. /etc/syslog

You may want to run the *syslog* program (to collect log information about sendmail). This program normally resides in */etc/syslog*, with support files */etc/syslog.conf* and */etc/syslog.pid*. The program is located in the *aux* subdirectory of the *sendmail* distribution. The file */etc/syslog.conf* describes the file(s) that sendmail will log in. For a complete description of syslog, see the manual page for *syslog*(8) (located in *sendmail/doc* on the distribution).

### 1.3.12. /usr/ucb/newaliases

If *sendmail* is invoked as "newaliases," it will simulate the **-bi** flag (i.e., will rebuild the alias database; see below). This should be a link to /usr/lib/sendmail.

### 1.3.13. /usr/ucb/mailq

If *sendmail* is invoked as "mailq," it will simulate the **-bp** flag (i.e., *sendmail* will print the contents of the mail queue; see below). This should be a link to /usr/lib/sendmail.

# 2. NORMAL OPERATIONS

## 2.1. Quick Configuration Startup

A fast version of the configuration file may be set up by using the **–bz** flag:

/usr/lib/sendmail –bz

This creates the file */usr/lib/sendmail.fc* ("frozen configuration"). This file is an image of *sendmail*'s data space after reading in the configuration file. If this file exists, it is used instead of */usr/lib/sendmail.cf* *sendmail.fc* must be rebuilt manually every time *sendmail.cf* is changed.

The frozen configuration file will be ignored if a **–C** flag is specified or if sendmail detects that it is out of date. However, the heuristics are not strong so this should not be trusted.

## 2.2. The System Log

The system log is supported by the *syslog*(8) program.

### 2.2.1. Format

Each line in the system log consists of a timestamp, the name of the machine that generated it (for logging from several machines over the ethernet), the word "sendmail:", and a message.

### 2.2.2. Levels

If you have *syslog*(8) or an equivalent installed, you will be able to do logging. There is a large amount of information that can be logged. The log is arranged as a succession of levels. At the lowest level only extremely strange situations are logged. At the highest level, even the most mundane and uninteresting events are recorded for posterity. As a convention, log levels under ten are considered "useful;" log levels above ten are usually for debugging purposes.

A complete description of the log levels is given in section 4.3.

## 2.3. The Mail Queue

The mail queue should be processed transparently. However, you may find that manual intervention is sometimes necessary. For example, if a major host is down for a period of time the queue may become clogged. Although sendmail ought to recover gracefully when the host comes up, you may find performance unacceptably bad in the meantime.

### 2.3.1. Printing the queue

The contents of the queue can be printed using the *mailq* command (or by specifying the **–bp** flag to sendmail):

mailq

This will produce a listing of the queue id's, the size of the message, the date the message entered the queue, and the sender and recipients.

### 2.3.2. Format of queue files

All queue files have the form x*f*AA99999 where *AA99999* is the *id* for this file and the *x* is a type. The types are:

d The data file. The message body (excluding the header) is kept in this file.

l The lock file. If this file exists, the job is currently being processed, and a queue run will not process the file. For that reason, an extraneous lf file can cause a job to apparently disappear (it will not even time out!).

n This file is created when an id is being created. It is a separate file to insure that no mail can ever be destroyed due to a race condition. It should exist for no more than a few milliseconds at any given time.

q The queue control file. This file contains the information necessary to process the job.

t A temporary file. These are an image of the qf file when it is being rebuilt. It should be renamed to a qf file very quickly.

x A transcript file, existing during the life of a session showing everything that happens during that session.

The qf file is structured as a series of lines each beginning with a code letter. The lines are as follows:

D The name of the data file. There may only be one of these lines.

H A header definition. There may be any number of these lines. The order is important: they represent the order in the final message. These use the same syntax as header definitions in the configuration file.

R A recipient address. This will normally be completely aliased, but is actually realiased when the job is processed. There will be one line for each recipient.

S The sender address. There may only be one of these lines.

T The job creation time. This is used to compute when to time out the job.

P The current message priority. This is used to order the queue. Higher numbers mean lower priorities. The priority increases as the message sits in the queue. The initial priority depends on the message class and the size of the message.

M A message. This line is printed by the *mailq* command, and is generally used to store status information. It can contain any text.

As an example, the following is a queue file sent to "mckusick@calder" and "wnj":

```
DdfA13557
Seric
T404261372
P132
Rmckusick@calder
Rwnj
H?D?date: 23-Oct-82 15:49:32-PDT (Sat)
H?F?from: eric (Eric Allman)
H?x?full-name: Eric Allman
Hsubject: this is an example message
Hmessage-id: <8209232249.13557@UCBARPA.BERKELEY.ARPA>
Hreceived: by UCBARPA.BERKELEY.ARPA (3.227 [10/22/82])
        id A13557; 23-Oct-82 15:49:32-PDT (Sat)
Hphone: (415) 548-3211
HTo: mckusick@calder, wnj
```

This shows the name of the data file, the person who sent the message, the submission time (in seconds since January 1, 1970), the message priority, the message class, the recipients, and the headers for the message.

### 2.3.3. Forcing the queue

*Sendmail* should run the queue automatically at intervals. The algorithm is to read and sort the queue, and then to attempt to process all jobs in order. When it attempts to run the job, *sendmail* first checks to see if the job is locked. If so, it ignores the job.

There is no attempt to insure that only one queue processor exists at any time, since there is no guarantee that a job cannot take forever to process. Due to the locking algorithm, it is impossible for one job to freeze the queue. However, an uncooperative recipient host or a program recipient that never returns can accumulate many processes in your system. Unfortunately, there is no way to resolve this without violating the protocol.

In some cases, you may find that a major host going down for a couple of days may create a prohibitively large queue. This will result in *sendmail* spending an inordinate amount of time sorting the queue. This situation can be fixed by moving the queue to a temporary place and creating a new queue. The old queue can be run later when the offending host returns to service.

To do this, it is acceptable to move the entire queue directory:

```
cd /usr/spool
mv mqueue omqueue; mkdir mqueue; chmod 777 mqueue
```

You should then kill the existing daemon (since it will still be processing in the old queue directory) and create a new daemon.

To run the old mail queue, run the following command:

```
/usr/lib/sendmail -oQ/usr/spool/omqueue -q
```

The **-oQ** flag specifies an alternate queue directory and the **-q** flag says to just run every job in the queue. If you have a tendency toward voyeurism, you can use the **-v** flag to watch what is going on.

When the queue is finally emptied, you can remove the directory:

```
rmdir /usr/spool/omqueue
```

## 2.4. The Alias Database

The alias database exists in two forms. One is a text form, maintained in the file */usr/lib/aliases*. The aliases are of the form

```
name: name1, name2, ...
```

Only local names may be aliased; e.g.,

```
eric@mit-xx: eric@berkeley
```

will not have the desired effect. Aliases may be continued by starting any continuation lines with a space or a tab. Blank lines and lines beginning with a sharp sign ("#") are comments.

The second form is processed by the *dbm*(3) library. This form is in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag*. This is the form that *sendmail* actually uses to resolve aliases. This technique is used to improve performance.

### 2.4.1. Rebuilding the alias database

The DBM version of the database may be rebuilt explicitly by executing the command

```
newaliases
```

This is equivalent to giving *sendmail* the **–bi** flag:

/usr/lib/sendmail –bi

If the "D" option is specified in the configuration, *sendmail* will rebuild the alias database automatically if possible when it is out of date. The conditions under which it will do this are:

(1) The DBM version of the database is mode 666. -or-

(2) *Sendmail* is running setuid to root.

Auto-rebuild can be dangerous on heavily loaded machines with large alias files; if it might take more than five minutes to rebuild the database, there is a chance that several processes will start the rebuild process simultaneously.

### 2.4.2. Potential problems

There are a number of problems that can occur with the alias database. They all result from a *sendmail* process accessing the DBM version while it is only partially built. This can happen under two circumstances: One process accesses the database while another process is rebuilding it, or the process rebuilding the database dies (due to being killed or a system crash) before completing the rebuild.

Sendmail has two techniques to try to relieve these problems. First, it ignores interrupts while rebuilding the database; this avoids the problem of someone aborting the process leaving a partially rebuilt database. Second, at the end of the rebuild it adds an alias of the form

@: @

(which is not normally legal). Before sendmail will access the database, it checks to insure that this entry exists[1]. *Sendmail* will wait for this entry to appear, at which point it will force a rebuild itself[2].

### 2.4.3. List owners

If an error occurs on sending to a certain address, say "x", *sendmail* will look for an alias of the form "owner-x" *to receive the errors. This is typically useful for a mailing list where the submitter of the list has no control over the maintanence of the list itself; in this case the list maintainer would be the owner of the list. For example:*

*unix-wizards: eric@ucbarpa, wnj@monet, nosuchuser,*
*sam@matisse*
*owner-unix-wizards: eric@ucbarpa*

*would cause "eric@ucbarpa" to get the error that will occur when someone sends to unix-wizards due to the inclusion of "nosuchuser" on the list.*

## 2.5. Per-User Forwarding (.forward Files)

As an alternative to the alias database, any user may put a file with the name ".forward" in his or her home directory. If this file exists, *sendmail* redirects mail for that user to the list of addresses listed in the .forward file. For example, if the home directory for user "mckusick" has a .forward file with contents:

---

[1] The "a" option is required in the configuration for this action to occur. This should normally be specified unless you are running *delivermail* in parallel with *sendmail*.

[2] Note: the "D" option must be specified in the configuration file for this operation to occur. If the "D" option is not specified, a warning message is generated and *sendmail* continues.

mckusick@ernie
kirk@calder

then any mail arriving for "mckusick" will be redirected to the specified accounts.

### 2.6. Special Header Lines

Several header lines have special interpretations defined by the configuration file. Others have interpretations built into *sendmail* that cannot be changed without changing the code. These builtins are described here.

#### 2.6.1. Return-Receipt-To:

If this header is sent, a message will be sent to any specified addresses when the final delivery is complete. if the mailer has the **l** flag (local delivery) set in the mailer descriptor.

#### 2.6.2. Errors-To:

If errors occur anywhere during processing, this header will cause error messages to go to the listed addresses rather than to the sender. This is intended for mailing lists.

#### 2.6.3. Apparently-To:

If a message comes in with no recipients listed in the message (in a To:, Cc:, or Bcc: line) then *sendmail* will add an "Apparently-To:" header line for any recipients it is aware of. This is not put in as a standard recipient line to warn any recipients that the list is not complete.

At least one recipient line is required under RFC 822.

## 3. ARGUMENTS

The complete list of arguments to *sendmail* is described in detail in Appendix A. Some important arguments are described here.

### 3.1. Queue Interval

The amount of time between forking a process to run through the queue is defined by the **–q** flag. If you run in mode **f** or **a** this can be relatively large, since it will only be relevant when a host that was down comes back up. If you run in **q** mode it should be relatively short, since it defines the maximum amount of time that a message may sit in the queue.

### 3.2. Daemon Mode

If you allow incoming mail over an IPC connection, you should have a daemon running. This should be set by your */etc/rc* file using the **–bd** flag. The **–bd** flag and the **–q** flag may be combined in one call:

```
/usr/lib/sendmail –bd –q30m
```

### 3.3. Forcing the Queue

In some cases you may find that the queue has gotten clogged for some reason. You can force a queue run using the **–q** flag (with no value). It is entertaining to use the **–v** flag (verbose) when this is done to watch what happens:

```
usr/lib/sendmail –q –v
```

### 3.4. Debugging

There are a fairly large number of debug flags built into *sendmail*. Each debug flag has a number and a level, where higher levels means to print out more information. The convention is that levels greater than nine are "absurd," i.e., they print out so much information that you wouldn't normally want to see them except for debugging that particular piece of code. Debug flags are set using the **-d** option; the syntax is:

```
debug-flag:    -d debug-list
debug-list:    debug-option [ , debug-option ]
debug-option:  debug-range [ . debug-level ]
debug-range:   integer | integer - integer
debug-level:   integer
```

where spaces are for reading ease only. For example,

```
-d12          Set flag 12 to level 1
-d12.3        Set flag 12 to level 3
-d3-17        Set flags 3 through 17 to level 1
-d3-17.4      Set flags 3 through 17 to level 4
```

For a complete list of the available debug flags you will have to look at the code (they are too dynamic to keep this documentation up to date).

### 3.5. Trying a Different Configuration File

An alternative configuration file can be specified using the **-C** flag; for example,

```
/usr/lib/sendmail -Ctest.cf
```

uses the configuration file *test.cf* instead of the default */usr/lib/sendmail.cf*. If the **-C** flag has no value it defaults to *sendmail.cf* in the current directory.

### 3.6. Changing the Values of Options

Options can be overridden using the **-o** flag. For example,

```
/usr/lib/sendmail -oT2m
```

sets the T (timeout) option to two minutes for this run only.

## 4. TUNING

There are a number of configuration parameters you may want to change, depending on the requirements of your site. Most of these are set using an option in the configuration file. For example, the line "OT3d" sets option "T" to the value "3d" (three days).

### 4.1. Timeouts

All time intervals are set using a scaled syntax. For example, "10m" represents ten minutes, whereas "2h30m" represents two and a half hours. The full set of scales is:

```
s   seconds
m   minutes
h   hours
d   days
w   weeks
```

#### 4.1.1. Queue interval

The argument to the **-q** flag specifies how often a subdaemon will run the queue. This is typically set to between five minutes and one half hour.

#### 4.1.2. Read timeouts

It is possible to time out when reading the standard input or when reading from a remote SMTP server. Technically, this is not acceptable within the published protocols. However, it might be appropriate to set it to something large in certain environments (such as an hour). This will reduce the chance of large numbers of idle daemons piling up on your system. This timeout is set using the r option in the configuration file.

#### 4.1.3. Message timeouts

After sitting in the queue for a few days, a message will time out. This is to insure that at least the sender is aware of the inability to send a message. The timeout is typically set to three days. This timeout is set using the T option in the configuration file.

The time of submission is set in the queue, rather than the amount of time left until timeout. As a result, you can flush messages that have been hanging for a short period by running the queue with a short message timeout. For example,

```
/usr/lib/sendmail -oT1d -q
```

will run the queue and flush anything that is one day old.

### 4.2. Delivery Mode

There are a number of delivery modes that *sendmail* can operate in, set by the "d" configuration option. These modes specify how quickly mail will be delivered. Legal modes are:

- i deliver interactively (synchronously)
- b deliver in background (asynchronously)
- q queue only (don't deliver)

There are tradeoffs. Mode "i" passes the maximum amount of information to the sender, but is hardly ever necessary. Mode "q" puts the minimum load on your machine, but means that delivery may be delayed for up to the queue interval. Mode "b" is probably a good compromise. However, this mode can cause large numbers of processes if you have a mailer that takes a long time to deliver a message.

### 4.3. Log Level

The level of logging can be set for sendmail. The default using a standard configuration table is level 9. The levels are as follows:

- 0 No logging.
- 1 Major problems only.
- 2 Message collections and failed deliveries.
- 3 Successful deliveries.
- 4 Messages being defered (due to a host being down, etc.).
- 5 Normal message queueups.
- 6 Unusual but benign incidents, e.g., trying to process a locked queue file.
- 9 Log internal queue id to external message id mappings. This can be useful for tracing a message as it travels between several hosts.
- 12 Several messages that are basically only of interest when debugging.
- 16 Verbose information regarding the queue.

### 4.4. Load Limiting

*Sendmail* can be asked to queue (but not deliver) mail if the system load average gets too high using the **x** option. When the load average exceeds the value of the **x** option, the delivery mode is set to **q** (queue only) until the load drops.

For drastic cases, the **X** option defines a load average at which sendmail will refuse to connect network connections. Locally generated mail (including incoming UUCP mail) is still accepted.

### 4.5. File Modes

There are a number of files that may have a number of modes. The modes depend on what functionality you want and the level of security you require.

#### 4.5.1. To suid or not to suid?

*Sendmail* can safely be made setuid to root. At the point where it is about to *exec*(2) a mailer, it checks to see if the userid is zero; if so, it resets the userid and groupid to a default (set by the **u** and **g** options). (This can be overridden by setting the **S** flag to the mailer for mailers that are trusted and must be called as root.) However, this will cause mail processing to be accounted (using *sa*(8)) to root rather than to the user sending the mail.

#### 4.5.2. Temporary file modes

The mode of all temporary files that *sendmail* creates is determined by the "F" option. Reasonable values for this option are 0600 and 0644. If the more permissive mode is selected, it will not be necessary to run *sendmail* as root at all (even when running the queue).

#### 4.5.3. Should my alias database be writable?

At Berkeley we have the alias database (/usr/lib/aliases*) mode 666. There are some dangers inherent in this approach: any user can add him-/her-self to any list, or can "steal" any other user's mail. However, we have found users to be basically trustworthy, and the cost of having a read-only database greater than the expense of finding and eradicating the rare nasty person.

The database that *sendmail* actually used is represented by the two files *aliases.dir* and *aliases.pag* (both in /usr/lib). The mode on these files should match the mode on /usr/lib/aliases. If *aliases* is writable and the DBM files (*aliases.dir* and *aliases.pag*) are not, users will be unable to reflect their desired changes through to the actual database. However, if *aliases* is read-only and the DBM files are writable, a slightly sophisticated user can arrange to steal mail anyway.

If your DBM files are not writable by the world or you do not have auto-rebuild enabled (with the "D" option), then you must be careful to reconstruct the alias database each time you change the text version:

```
newaliases
```

If this step is ignored or forgotten any intended changes will also be ignored or forgotten.

## 5. THE WHOLE SCOOP ON THE CONFIGURATION FILE

This section describes the configuration file in detail, including hints on how to write one of your own if you have to.

There is one point that should be made clear immediately: the syntax of the configuration file is designed to be reasonably easy to parse, since this is done every time *sendmail* starts up, rather than easy for a human to read or write. On the "future project" list is a configuration-file compiler.

An overview of the configuration file is given first, followed by details of the semantics.

## 5.1. The Syntax

The configuration file is organized as a series of lines, each of which begins with a single character defining the semantics for the rest of the line. Lines beginning with a space or a tab are continuation lines (although the semantics are not well defined in many places). Blank lines and lines beginning with a sharp symbol ('#') are comments.

### 5.1.1. R and S – rewriting rules

The core of address parsing are the rewriting rules. These are an ordered production system. *Sendmail* scans through the set of rewriting rules looking for a match on the left hand side (LHS) of the rule. When a rule matches, the address is replaced by the right hand side (RHS) of the rule.

There are several sets of rewriting rules. Some of the rewriting sets are used internally and must have specific semantics. Other rewriting sets do not have specifically assigned semantics, and may be referenced by the mailer definitions or by other rewriting sets.

The syntax of these two commands are:

**S***n*

Sets the current ruleset being collected to *n*. If you begin a ruleset more than once it deletes the old definition.

**R***lhs rhs comments*

The fields must be separated by at least one tab character; there may be embedded spaces in the fields. The *lhs* is a pattern that is applied to the input. If it matches, the input is rewritten to the *rhs*. The *comments* are ignored.

### 5.1.2. D – define macro

Macros are named with a single character. These may be selected from the entire ASCII set, but user-defined macros should be selected from the set of upper case letters only. Lower case letters and special symbols are used internally.

The syntax for macro definitions is:

**D***x val*

where *x* is the name of the macro and *val* is the value it should have. Macros can be interpolated in most places using the escape sequence **$***x*.

### 5.1.3. C and F – define classes

Classes of words may be defined to match on the left hand side of rewriting rules. For example a class of all local names for this site might be created so that attempts to send to oneself can be eliminated. These can either be defined directly in the configuration file or read in from another file. Classes may be given names from the set of upper case letters. Lower case letters and special characters are reserved for system use.

The syntax is:

**C***c word1 word2...*
**F***c file* [ *format* ]

The first form defines the class *c* to match any of the named words. It is permissible to split them among multiple lines; for example, the two forms:

```
CHmonet ucbmonet
```

and

```
CHmonet
CHucbmonet
```

are equivalent. The second form reads the elements of the class *c* from the named *file*; the *format* is a *scanf*(3) pattern that should produce a single string.

### 5.1.4. M – define mailer

Programs and interfaces to mailers are defined in this line. The format is:

**M***name, {field=value}**

where *name* is the name of the mailer (used internally only) and the "field=name" pairs define attributes of the mailer. Fields are:

| | |
|---|---|
| Path | The pathname of the mailer |
| Flags | Special flags for this mailer |
| Sender | A rewriting set for sender addresses |
| Recipient | A rewriting set for recipient addresses |
| Argv | An argument vector to pass to this mailer |
| Eol | The end-of-line string for this mailer |
| Maxsize | The maximum message length to this mailer |

Only the first character of the field name is checked.

### 5.1.5. H – define header

The format of the header lines that sendmail inserts into the message are defined by the H line. The syntax of this line is:

**H**[*?mflags?*]*hname*: *htemplate*

Continuation lines in this spec are reflected directly into the outgoing message. The *htemplate* is macro expanded before insertion into the message. If the *mflags* (surrounded by question marks) are specified, at least one of the specified flags must be stated in the mailer definition for this header to be automatically output. If one of these headers is in the input it is reflected to the output regardless of these flags.

Some headers have special semantics that will be described below.

### 5.1.6. O – set option

There are a number of "random" options that can be set from a configuration file. Options are represented by single characters. The syntax of this line is:

**O***o value*

This sets option *o* to be *value*. Depending on the option, *value* may be a string, an integer, a boolean (with legal values "t", "T", "f", or "F"; the default is TRUE), or a time interval.

#### 5.1.7. T – define trusted users

Trusted users are those users who are permitted to override the sender address using the **–f** flag. These typically are "root," "uucp," and "network," but on some users it may be convenient to extend this list to include other users, perhaps to support a separate UUCP login for each host. The syntax of this line is:

T*user1 user2...*

There may be more than one of these lines.

#### 5.1.8. P – precedence definitions

Values for the "Precedence:" field may be defined using the **P** control line. The syntax of this field is:

Pname=num

When the *name* is found in a "Precedence:" field, the message class is set to *num*. Higher numbers mean higher precedence. Numbers less than zero have the special property that error messages will not be returned. The default precedence is zero. For example, our list of precedences is:

```
Pfirst-class=0
Pspecial-delivery=100
Pjunk=-100
```

### 5.2. The Semantics

This section describes the semantics of the configuration file.

#### 5.2.1. Special macros, conditionals

Macros are interpolated using the construct **$***x*, where *x* is the name of the macro to be interpolated. In particular, lower case letters are reserved to have special semantics, used to pass information in or out of sendmail, and some special characters are reserved to provide conditionals, etc.

The following macros *must* be defined to transmit information into *sendmail:*

- e The SMTP entry message
- j The "official" domain name for this site
- l The format of the UNIX from line
- n The name of the daemon (for error messages)
- o The set of "operators" in addresses
- q default format of sender address

The **$e** macro is printed out when SMTP starts up. The first word must be the **$j** macro. The **$j** macro should be in RFC821 format. The **$l** and **$n** macros can be considered constants except under terribly unusual circumstances. The **$o** macro consists of a list of characters which will be considered tokens and which will separate tokens when doing parsing. For example, if "r" were in the **$o** macro, then the input "address" would be scanned as three tokens: "add," "r," and "ess." Finally, the **$q** macro specifies how an address should appear in a message when it is defaulted. For example, on our system these definitions are:

```
De$j Sendmail $v ready at $b
DnMAILER-DAEMON
DlFrom $g $d
Do.:%@!^=/
Dq$g$?x ($x)$.
Dj$H.$D
```

An acceptable alternative for the **$q** macro is "$?x$x $.<$g>". These correspond to the following two formats:

```
eric@Berkeley (Eric Allman)
Eric Allman <eric@Berkeley>
```

Some macros are defined by *sendmail* for interpolation into argv's for mailers or for other contexts. These macros are:

| | |
|---|---|
| a | The origination date in Arpanet format |
| b | The current date in Arpanet format |
| c | The hop count |
| d | The date in UNIX (ctime) format |
| f | The sender (from) address |
| g | The sender address relative to the recipient |
| h | The recipient host |
| i | The queue id |
| p | Sendmail's pid |
| r | Protocol used |
| s | Sender's host name |
| t | A numeric representation of the current time |
| u | The recipient user |
| v | The version number of sendmail |
| w | The hostname of this site |
| x | The full name of the sender |
| z | The home directory of the recipient |

There are three types of dates that can be used. The **$a** and **$b** macros are in Arpanet format; **$a** is the time as extracted from the "Date:" line of the message (if there was one), and **$b** is the current date and time (used for postmarks). If no "Date:" line is found in the incoming message, **$a** is set to the current time also. The **$d** macro is equivalent to the **$a** macro in UNIX (ctime) format.

The **$f** macro is the id of the sender as originally determined; when mailing to a specific host the **$g** macro is set to the address of the sender *relative to the recipient*. For example, if I send to "bollard@matisse" from the machine "ucbarpa" the **$f** macro will be "eric" and the **$g** macro will be "eric@ucbarpa."

The **$x** macro is set to the full name of the sender. This can be determined in several ways. It can be passed as flag to *sendmail*. The second choice is the value of the "Full-name:" line in the header if it exists, and the third choice is the comment field of a "From:" line. If all of these fail, and if the message is being originated locally, the full name is looked up in the */etc/passwd* file.

When sending, the **$h**, **$u**, and **$z** macros get set to the host, user, and home directory (if local) of the recipient. The first two are set from the **$@** and **$:** part of the rewriting rules, respectively.

The **$p** and **$t** macros are used to create unique strings (e.g., for the "Message-Id:" field). The **$i** macro is set to the queue id on this host; if put into the timestamp line it can be extremely useful for tracking messages. The **$v** macro

is set to be the version number of *sendmail*; this is normally put in timestamps and has been proven extremely useful for debugging. The **$w** macro is set to the name of this host if it can be determined. The **$c** field is set to the "hop count," i.e., the number of times this message has been processed. This can be determined by the **-h** flag on the command line or by counting the timestamps in the message.

The **$r** and **$s** fields are set to the protocol used to communicate with sendmail and the sending hostname; these are not supported in the current version.

Conditionals can be specified using the syntax:

$?x text1 $| text2 $.

This interpolates *text1* if the macro **$x** is set, and *text2* otherwise. The "else" (**$|**) clause may be omitted.

### 5.2.2. Special classes

The class **$=w** is set to be the set of all names this host is known by. This can be used to delete local hostnames.

### 5.2.3. The left hand side

The left hand side of rewriting rules contains a pattern. Normal words are simply matched directly. Metasyntax is introduced using a dollar sign. The metasymbols are:

| | |
|---|---|
| $* | Match zero or more tokens |
| $+ | Match one or more tokens |
| $- | Match exactly one token |
| $=x | Match any token in class x |
| $~x | Match any token not in class x |

If any of these match, they are assigned to the symbol **$***n* for replacement on the right hand side, where *n* is the index in the LHS. For example, if the LHS:

$-:$+

is applied to the input:

UCBARPA:eric

the rule will match, and the values passed to the RHS will be:

$1 UCBARPA
$2 eric

### 5.2.4. The right hand side

When the left hand side of a rewriting rule matches, the input is deleted and replaced by the right hand side. Tokens are copied directly from the RHS unless they are begin with a dollar sign. Metasymbols are:

| | |
|---|---|
| $n | Substitute indefinite token n from LHS |
| $[name$] | Canonicalize name |
| $>n | "Call" ruleset n |
| $#mailer | Resolve to mailer |
| $@host | Specify host |
| $:user | Specify user |

The **$***n* syntax substitutes the corresponding value from a **$+**, **$-**, **$***, **$=**, or **$~** match on the LHS. It may be used anywhere.

A host name enclosed between **$[** and **$]** is looked up in the */etc/hosts* file and replaced by the canonical name. For example, "$[csam$]" would become "lbl-csam.arpa."

The **$>***n* syntax causes the remainder of the line to be substituted as usual and then passed as the argument to ruleset *n*. The final value of ruleset *n* then becomes the substitution for this rule.

The **$#** syntax should *only* be used in ruleset zero. It causes evaluation of the ruleset to terminate immediately, and signals to sendmail that the address has completely resolved. The complete syntax is:

$#mailer$@host$:user

This specifies the {mailer, host, user} 3-tuple necessary to direct the mailer. If the mailer is local the host part may be omitted. The *mailer* and *host* must be a single word, but the *user* may be multi-part.

A RHS may also be preceeded by a **$@** or a **$:** to control evaluation. A **$@** prefix causes the ruleset to return with the remainder of the RHS as the value. A **$:** prefix causes the rule to terminate immediately, but the ruleset to continue; this can be used to avoid continued application of a rule. The prefix is stripped before continuing.

The **$@** and **$:** prefixes may preceed a **$>** spec; for example:

R$+ $:$>7$1

matches anything, passes that to ruleset seven, and continues; the **$:** is necessary to avoid an infinite loop.

Substitution occurs in the order described, that is, parameters from the LHS are substituted, hostnames are canonicalized, "subroutines" are called, and finally **$#**, **$@**, and **$:** are processed.

### 5.2.5. Semantics of rewriting rule sets

There are five rewriting sets that have specific semantics. These are related as depicted by figure 2.

Ruleset three should turn the address into "canonical form." This form should have the basic syntax:

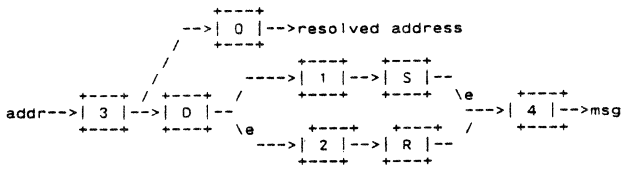



Figure 2-1 Rewriting set semantics

local-part@host-domain-spec

If no "@" sign is specified, then the host-domain-spec *may* be appended from the sender address (if the C flag is set in the mailer definition corresponding to the *sending* mailer). Ruleset three is applied by sendmail before doing anything with any address.

Ruleset zero is applied after ruleset three to addresses that are going to actually specify recipients. It must resolve to a {*mailer, host, user*} triple. The *mailer* must be defined in the mailer definitions from the configuration file. The *host* is defined into the **$h** macro for use in the argv expansion of the specified mailer.

Rulesets one and two are applied to all sender and recipient addresses respectively. They are applied before any specification in the mailer definition. They must never resolve.

Ruleset four is applied to all addresses in the message. It is typically used to translate internal to external form.

### 5.2.6. Mailer flags etc.

There are a number of flags that may be associated with each mailer, each identified by a letter of the alphabet. Many of them are assigned semantics internally. These are detailed in Appendix C. Any other flags may be used freely to conditionally assign headers to messages destined for particular mailers.

### 5.2.7. The "error" mailer

The mailer with the special name "error" can be used to generate a user error. The (optional) host field is a numeric exit status to be returned, and the user field is a message to be printed. For example, the entry:

$#error$:Host unknown in this domain

on the RHS of a rule will cause the specified error to be generated if the LHS matches. This mailer is only functional in ruleset zero.

## 5.3. Building a Configuration File From Scratch

Building a configuration table from scratch is an extremely difficult job. Fortunately, it is almost never necessary to do so; nearly every situation that may come up may be resolved by changing an existing table. In any case, it is critical that you understand what it is that you are trying to do and come up with a philosophy for the configuration table. This section is intended to explain what the real purpose of a configuration table is and to give you some ideas for what your philosophy might be.

### 5.3.1. What you are trying to do

The configuration table has three major purposes. The first and simplest is to set up the environment for *sendmail*. This involves setting the options, defining a few critical macros, etc. Since these are described in other places, we will not go into more detail here.

The second purpose is to rewrite addresses in the message. This should typically be done in two phases. The first phase maps addresses in any format into a canonical form. This should be done in ruleset three. The second phase maps this canonical form into the syntax appropriate for the receiving mailer. *Sendmail* does this in three subphases. Rulesets one and two are applied to all sender and recipient addresses respectively. After this, you may specify per-mailer rulesets for both sender and recipient addresses; this allows mailer-specific customization.

Finally, ruleset four is applied to do any default conversion to external form.

The third purpose is to map addresses into the actual set of instructions necessary to get the message delivered. Ruleset zero must resolve to the internal form, which is in turn used as a pointer to a mailer descriptor. The mailer descriptor describes the interface requirements of the mailer.

### 5.3.2. Philosophy

The particular philosophy you choose will depend heavily on the size and structure of your organization. I will present a few possible philosophies here.

One general point applies to all of these philosophies: it is almost always a mistake to try to do full name resolution. For example, if you are trying to get names of the form "user@host" to the Arpanet, it does not pay to route them to "xyzvax!decvax!ucbvax!c70:user@host" since you then depend on several links not under your control. The best approach to this problem is to simply forward to "xyzvax!user@host" and let xyzvax worry about it from there. In summary, just get the message closer to the destination, rather than determining the full path.

#### 5.3.2.1. Large site, many hosts – minimum information

Berkeley is an example of a large site, i.e., more than two or three hosts. We have decided that the only reasonable philosophy in our environment is to designate one host as the guru for our site. It must be able to resolve any piece of mail it receives. The other sites should have the minimum amount of information they can get away with. In addition, any information they do have should be hints rather than solid information.

. For example, a typical site on our local ether network is "monet." Monet has a list of known ethernet hosts; if it receives mail for any of them, it can do direct delivery. If it receives mail for any unknown host, it just passes it directly to "ucbvax," our master host. Ucbvax may determine that the host name is illegal and reject the message, or may be able to do delivery. However, it is important to note that when a new ethernet host is added, the only host that *must* have its tables updated is ucbvax; the others *may* be updated as convenient, but this is not critical.

This picture is slightly muddied due to network connections that are not actually located on ucbvax. For example, our TCP connection is currently on "ucbarpa." However, monet *does not* know about this; the information is hidden totally between ucbvax and ucbarpa. Mail going from monet to a TCP host is transfered via the ethernet from monet to ucbvax, then via the ethernet from ucbvax to ucbarpa, and then is submitted to the Arpanet. Although this involves some extra hops, we feel this is an acceptable tradeoff.

An interesting point is that it would be possible to update monet to send TCP mail directly to ucbarpa if the load got too high; if monet failed to note a host as a TCP host it would go via ucbvax as before, and if monet incorrectly sent a message to ucbarpa it would still be sent by ucbarpa to ucbvax as before. The only problem that can occur is loops, as if ucbarpa thought that ucbvax had the TCP connection and vice versa. For this reason, updates should *always* happen to the master host first.

This philosophy results as much from the need to have a single source for the configuration files (typically built using *m4*(1) or some similar tool) as any logical need. Maintaining more than three separate tables by hand is essentially an impossible job.

#### 5.3.2.2. Small site – complete information

A small site (two or three hosts) may find it more reasonable to have complete information at each host. This would require that each host know exactly where each network connection is, possibly including the names of each host on that network. As long as the site remains small and the the configuration remains relatively static, the update problem will probably not be too great.

#### 5.3.2.3. Single host

This is in some sense the trivial case. The only major issue is trying to insure that you don't have to know too much about your environment. For example, if you have a UUCP connection you might find it useful to know about the names of hosts connected directly to you, but this is really not necessary since this may be determined from the syntax.

### 5.3.3. Relevant issues

The canonical form you use should almost certainly be as specified in the Arpanet protocols RFC819 and RFC822. Copies of these RFC's are included on the *sendmail* tape as *doc/rfc819.lpr* and *doc/rfc822.lpr*.

RFC822 describes the format of the mail message itself. *Sendmail* follows this RFC closely, to the extent that many of the standards described in this document can not be changed without changing the code. In particular, the following characters have special interpretations:

< > ( ) " \

Any attempt to use these characters for other than their RFC822 purpose in addresses is probably doomed to disaster.

RFC819 describes the specifics of the domain-based addressing. This is touched on in RFC822 as well. Essentially each host is given a name which is a right-to-left dot qualified pseudo-path from a distinguished root. The elements of the path need not be physical hosts; the domain is logical rather than physical. For example, at Berkeley one legal host is "a.cc.berkeley.arpa"; reading from right to left, "arpa" is a top level domain (related to, but not limited to, the physical Arpanet), "berkeley" is both an Arpanet host and a logical domain which is actually interpreted by a host called ucbvax (which is actually just the "major" host for this domain), "cc" represents the Computer Center, (in this case a strictly logical entity), and "a" is a host in the Computer Center; this particular host happens to be connected via berknet, but other hosts might be connected via one of two ethernets or some other network.

Beware when reading RFC819 that there are a number of errors in it.

### 5.3.4. How to proceed

Once you have decided on a philosophy, it is worth examining the available configuration tables to decide if any of them are close enough to steal major parts of. Even under the worst of conditions, there is a fair amount of boiler plate that can be collected safely.

The next step is to build ruleset three. This will be the hardest part of the job. Beware of doing too much to the address in this ruleset, since anything you do will reflect through to the message. In particular, stripping of local domains is best deferred, since this can leave you with addresses with no domain spec at all. Since *sendmail* likes to append the sending domain to addresses with no domain, this can change the semantics of addresses. Also try to avoid fully qualifying domains in

this ruleset. Although technically legal, this can lead to unpleasantly and unnecessarily long addresses reflected into messages. The Berkeley configuration files define ruleset nine to qualify domain names and strip local domains. This is called from ruleset zero to get all addresses into a cleaner form.

Once you have ruleset three finished, the other rulesets should be relatively trivial. If you need hints, examine the supplied configuration tables.

### 5.3.5. Testing the rewriting rules – the –bt flag

When you build a configuration table, you can do a certain amount of testing using the "test mode" of *sendmail*. For example, you could invoke *sendmail* as:

sendmail –bt –Ctest.cf

which would read the configuration file "test.ct" and enter test mode. In this mode, you enter lines of the form:

rwset address

where *rwset* is the rewriting set you want to use and *address* is an address to apply the set to. Test mode shows you the steps it takes as it proceeds, finally showing you the address it ends up with. You may use a comma separated list of rwsets for sequential application of rules to an input; ruleset three is always applied first. For example:

1,21,4 monet:bollard

first applies ruleset three to the input "monet:bollard." Ruleset one is then applied to the output of ruleset three, followed similarly by rulesets twenty-one and four.

If you need more detail, you can also use the "–d21" flag to turn on more debugging. For example,

sendmail –bt –d21.99

turns on an incredible amount of information; a single word address is probably going to print out several pages worth of information.

### 5.3.6. Building mailer descriptions

To add an outgoing mailer to your mail system, you will have to define the characteristics of the mailer.

Each mailer must have an internal name. This can be arbitrary, except that the names "local" and "prog" must be defined.

The pathname of the mailer must be given in the P field. If this mailer should be accessed via an IPC connection, use the string "[IPC]" instead.

The F field defines the mailer flags. You should specify an "f" or "r" flag to pass the name of the sender as a –f or –r flag respectively. These flags are only passed if they were passed to *sendmail*, so that mailers that give errors under some circumstances can be placated. If the mailer is not picky you can just specify "–f $g" in the argv template. If the mailer must be called as **root** the "S" flag should be given; this will not reset the userid before calling the mailer[3]. If this mailer is local (i.e., will perform final delivery rather than another network hop) the "l" flag should be given. Quote characters (backslashes and " marks) can be stripped from addresses if the "s" flag is specified; if this is not given they are passed through. If the mailer is capable of sending to more than one user on the same host in a single transaction the "m" flag should be stated. If this flag is on, then the argv template

[3] *Sendmail* must be running setuid to root for this to work.

containing **$u** will be repeated for each unique user on a given host. The "e" flag will mark the mailer as being "expensive," which will cause *sendmail* to defer connection until a queue run[1].

An unusual case is the "C" flag. This flag applies to the mailer that the message is received from, rather than the mailer being sent to; if set, the domain spec of the sender (i.e., the "@host.domain" part) is saved and is appended to any addresses in the message that do not already contain a domain spec. For example, a message of the form:

From: eric@ucbarpa
To: wnj@monet, mckusick

will be modified to:

From: eric@ucbarpa
To: wnj@monet, mckusick@ucbarpa

*if and only if* the "C" flag is defined in the mailer corresponding to "eric@ucbarpa."

Other flags are described in Appendix C.

The S and R fields in the mailer description are per-mailer rewriting sets to be applied to sender and recipient addresses respectively. These are applied after the sending domain is appended and the general rewriting sets (numbers one and two) are applied, but before the output rewrite (ruleset four) is applied. A typical use is to append the current domain to addresses that do not already have a domain. For example, a header of the form:

From: eric

might be changed to be:

From: eric@ucbarpa

or

From: ucbvax!eric

depending on the domain it is being shipped into. These sets can also be used to do special purpose output rewriting in cooperation with ruleset four.

The E field defines the string to use as an end-of-line indication. A string containing only newline is the default. The usual backslash escapes (\r, \n, \f, \b) may be used.

Finally, an argv template is given as the E field. It may have embedded spaces. If there is no argv with a **$u** macro in it, *sendmail* will speak SMTP to the mailer. If the pathname for this mailer is "[IPC]," the argv should be

IPC $h [ port ]

where *port* is the optional port number to connect to.

For example, the specifications:

Mlocal, P=/bin/mail, F=rlsm S=10, R=20, A=mail -d $u
Mether, P=[IPC], F=meC, S=11, R=21, A=IPC $h, M=100000

specifies a mailer to do local delivery and a mailer for ethernet delivery. The first is called "local," is located in the file "/bin/mail," takes a picky -r flag, does local delivery, quotes should be stripped from addresses, and multiple users can be delivered at once; ruleset ten should be applied to sender addresses in the message and ruleset twenty should be applied to recipient addresses; the argv to send to a

[1] The "c" configuration option must be given for this to be effective.

message will be the word "mail," the word "-d," and words containing the name of the receiving user. If a -r flag is inserted it will be between the words "mail" and "-d." The second mailer is called "ether," it should be connected to via an IPC connection, it can handle multiple users at once, connections should be deferred, and any domain from the sender address should be appended to any receiver name without a domain; sender addresses should be processed by ruleset eleven and recipient addresses by ruleset twenty-one. There is a 100,000 byte limit on messages passed through this mailer.

# APPENDIX A

# COMMAND LINE FLAGS

Arguments must be presented with flags before addresses. The flags are:

-f addr — The sender's machine address is *addr*. This flag is ignored unless the real user is listed as a "trusted user" or if *addr* contains an exclamation point (because of certain restrictions in UUCP).

-r addr — An obsolete form of -f.

-h cnt — Sets the "hop count" to *cnt*. This represents the number of times this message has been processed by *sendmail* (to the extent that it is supported by the underlying networks). *Cnt* is incremented during processing, and if it reaches MAXHOP (currently 30) *sendmail* throws away the message with an error.

-Fname — Sets the full name of this user to *name*.

-n — Don't do aliasing or forwarding.

-t — Read the header for "To:", "Cc:", and "Bcc:" lines, and send to everyone listed in those lists. The "Bcc:" line will be deleted before sending. Any addresses in the argument vector will be deleted from the send list.

-bx — Set operation mode to *x*. Operation modes are:

- m Deliver mail (default)
- a Run in arpanet mode (see below)
- s Speak SMTP on input side
- d Run as a daemon
- t Run in test mode
- v Just verify addresses, don't collect or deliver
- i Initialize the alias database
- p Print the mail queue
- z Freeze the configuration file

The special processing for the ARPANET includes reading the "From:" line from the header to find the sender, printing ARPANET style messages (preceded by three digit reply codes for compatibility with the FTP protocol [Neigus73, Postel74, Postel77]), and ending lines of error messages with <CRLF>.

-qtime — Try to process the queued up mail. If the time is given, a sendmail will run through the queue at the specified interval to deliver queued mail; otherwise, it only runs once.

-Cfile — Use a different configuration file. *Sendmail* runs as the invoking user (rather than root) when this flag is specified.

-dlevel — Set debugging level.

-ox value — Set option *x* to the specified *value*. These options are described in Appendix B.

There are a number of options that may be specified as primitive flags (provided for compatibility with *delivermail*). These are the e, i, m, and v options. Also, the f option may be specified as the -s flag.

# APPENDIX B

# CONFIGURATION OPTIONS

The following options may be set using the **–o** flag on the command line or the **O** line in the configuration file. Many of them cannot be specified unless the invoking user is trusted.

Afile — Use the named *file* as the alias file. If no file is specified, use *aliases* in the current directory.

aN — If set, wait up to *N* minutes for an "@:@" entry to exist in the alias database before starting up. If it does not appear in *N* minutes, rebuild the database (if the **D** option is also set) or issue a warning.

Bc — Set the blank substitution character to *c*. Unquoted spaces in addresses are replaced by this character.

c — If an outgoing mailer is marked as being expensive, don't connect immediately. This requires that queueing be compiled in, since it will depend on a queue run process to actually send the mail.

dx — Deliver in mode *x*. Legal modes are:

- i Deliver interactively (synchronously)
- b Deliver in background (asynchronously)
- q Just queue the message (deliver during queue run)

D — If set, rebuild the alias database if necessary and possible. If this option is not set, *sendmail* will never rebuild the alias database unless explicitly requested using **–bi**.

ex — Dispose of errors using mode *x*. The values for *x* are:

- p Print error messages (default)
- q No messages, just give exit status
- m Mail back errors
- w Write back errors (mail if user not logged in)
- e Mail back errors and give zero exit stat always

Fn — The temporary file mode, in octal. 644 and 600 are good choices.

f — Save Unix-style "From" lines at the front of headers. Normally they are assumed redundant and discarded.

gn — Set the default group id for mailers to run in to *n*.

Hfile — Specify the help file for SMTP.

i — Ignore dots in incoming messages.

Ln — Set the default log level to *n*.

Mx value — Set the macro *x* to *value*. This is intended only for use from the command line.

m — Send to me too, even if I am in an alias expansion.

Nnetname — The name of the home network; "ARPA" by default. The the argument of an SMTP "HELO" command is checked against "hostname.netname" where *hostname* is requested from the kernel for the current connection. If they do

| | |
|---|---|
| | not match, "Received:" lines are augmented by the name that is determined in this manner so that messages can be traced accurately. |
| o | Assume that the headers may be in old format, i.e., spaces delimit names. This actually turns on an adaptive algorithm: if any recipient address contains a comma, parenthesis, or angle bracket, it will be assumed that commas already exist. If this flag is not on, only commas delimit names. Headers are always output with commas between the names. |
| Qdir | Use the named *dir* as the queue directory. |
| qfactor | Use *factor* as the multiplier in the map function to decide when to just queue up jobs rather than run them. This value is divided by the difference between the current load average and the load average limit (x flag) to determine the maximum message priority that will be sent. Defaults to 10000. |
| rtime | Timeout reads after *time* interval. |
| Sfile | Log statistics in the named *file*. |
| s | Be super-safe when running things, i.e., always instantiate the queue file, even if you are going to attempt immediate delivery. *Sendmail* always instantiates the queue file before returning control the the client under any circumstances. |
| Ttime | Set the queue timeout to *time*. After this interval, messages that have not been successfully sent will be returned to the sender. |
| tS,D | Set the local timezone name to *S* for standard time and *D* for daylight time; this is only used under version six. |
| un | Set the default userid for mailers to *n*. Mailers without the *S* flag in the mailer definition will run as this user. |
| v | Run in verbose mode. |
| xLA | When the system load average exceeds *LA*, just queue messages (i.e., don't try to send them). |
| XLA | When the system load average exceeds *LA*, refuse incoming SMTP connections. |
| z | If set, deliver each job that is run from the queue in a separate process. Use this option if you are short of memory, since the default tends to consume considerable amounts of memory while the queue is being processed. |

# APPENDIX C

# MAILER FLAGS

The following flags may be set in the mailer description.

f The mailer wants a **-f** *from* flag, but only if this is a network forward operation (i.e., the mailer will give an error if the executing user does not have special permissions).

r Same as **f**, but sends a **-r** flag.

S Don't reset the userid before calling the mailer. This would be used in a secure environment where *sendmail* ran as root. This could be used to avoid forged addresses. This flag is suppressed if given from an "unsafe" environment (e.g, a user's mail.cf file).

n Do not insert a UNIX-style "From" line on the front of the message.

l This mailer is local (i.e., final delivery will be performed).

s Strip quote characters off of the address before calling the mailer.

m This mailer can send to multiple users on the same host in one transaction. When a **$u** macro occurs in the *argv* part of the mailer definition, that field will be repeated as necessary for all qualifying users.

F This mailer wants a "From:" header line.

D This mailer wants a "Date:" header line.

M This mailer wants a "Message-Id:" header line.

x This mailer wants a "Full-Name:" header line.

P This mailer wants a "Return-Path:" line.

u Upper case should be preserved in user names for this mailer.

h Upper case should be preserved in host names for this mailer.

A This is an Arpanet-compatible mailer, and all appropriate modes should be set.

U This mailer wants Unix-style "From" lines with the ugly UUCP-style "remote from <host>" on the end.

e This mailer is expensive to connect to, so try to avoid connecting normally; any necessary connection will occur during a queue run.

X This mailer want to use the hidden dot algorithm as specified in RFC821; basically, any line beginning with a dot will have an extra dot prepended (to be stripped at the other end). This insures that lines in the message containing a dot will not terminate the message prematurely.

L Limit the line lengths as specified in RFC821.

P Use the return-path in the SMTP "MAIL FROM:" command rather than just the return address; although this is required in RFC821, many hosts do not process return paths properly.

I This mailer will be speaking SMTP to another *sendmail* - as such it can use special protocol features. This option is not required (i.e., if this option is omitted the transmission will still operate successfully, although perhaps not as efficiently as possible).

C If mail is *received* from a mailer with this flag set, any addresses in the header that do not have an at sign ("@") after being rewritten by ruleset three will have the "@domain"

clause from the sender tacked on. This allows mail with headers of the form:

From: usera@hosta
To: userb@hostb, userc

to be rewritten as:

From: usera@hosta
To: userb@hostb, userc@hosta

automatically.

E Escape lines beginning with "From" in the message with a '>' sign.

# APPENDIX D

# OTHER CONFIGURATION

There are some configuration changes that can be made by recompiling *sendmail*. These are located in three places:

md/config.m4 These contain operating-system dependent descriptions. They are interpolated into the Makefiles in the *src* and *aux* directories. This includes information about what version of UNIX you are running, what libraries you have to include, etc.

src/conf.h Configuration parameters that may be tweaked by the installer are included in conf.h.

src/conf.c Some special routines and a few variables may be defined in conf.c. For the most part these are selected from the settings in conf.h.

**Parameters in md/config.m4**

The following compilation flags may be defined in the *m4CONFIG* macro in *md/config.m4* to define the environment in which you are operating.

V6 If set, this will compile a version 6 system, with 8-bit user id's, single character tty id's, etc.

VMUNIX If set, you will be assumed to have a Berkeley 4BSD or 4.1BSD, including the *vfork*(2) system call, special types defined in <sys/types.h> (e.g, u_char), etc.

If none of these flags are set, a version 7 system is assumed.

You will also have to specify what libraries to link with *sendmail* in the *m4LIBS* macro. Most notably, you will have to include it you are running a 4.1BSD system.

**Parameters in src/conf.h**

Parameters and compilation options are defined in conf.h. Most of these need not normally be tweaked; common parameters are all in sendmail.cf. However, the sizes of certain primitive vectors, etc., are included in this file. The numbers following the parameters are their default value.

MAXLINE [256] The maximum line length of any input line. If message lines exceed this length they will still be processed correctly; however, header lines, configuration file lines, alias lines, etc., must fit within this limit.

MAXNAME [128] The maximum length of any name, such as a host or a user name.

MAXFIELD [2500] The maximum total length of any header field, including continuation lines.

MAXPV [40] The maximum number of parameters to any mailer. This limits the number of recipients that may be passed in one transaction.

MAXHOP [30] When a message has been processed more than this number of times, sendmail rejects the message on the assumption that there has been an aliasing loop. This can be determined from the **-h** flag or by counting the number of trace fields (i.e, "Received:" lines) in the message header.

MAXATOM [100] The maximum number of atoms (tokens) in a single address. For example, the address "eric@Berkeley" is three atoms.

MAXMAILERS [25]
The maximum number of mailers that may be defined in the configuration file.

MAXRWSETS [30] The maximum number of rewriting sets that may be defined.

MAXPRIORITIES [25]
The maximum number of values for the "Precedence:" field that may be defined (using the **P** line in sendmail.cf).

MAXTRUST [30] The maximum number of trusted users that may be defined (using the **T** line in sendmail.cf).

A number of other compilation options exist. These specify whether or not specific code should be compiled in.

DBM If set, the "DBM" package in UNIX is used (see DBM(3X) in [UNIX80]). If not set, a much less efficient algorithm for processing aliases is used.

DEBUG If set, debugging information is compiled in. To actually get the debugging output, the **-d** flag must be used.

LOG If set, the *syslog* routine in use at some sites is used. This makes an informational log record for each message processed, and makes a higher priority log record for internal system errors.

QUEUE This flag should be set to compile in the queueing code. If this is not set, mailers must accept the mail immediately or it will be returned to the sender.

SMTP If set, the code to handle user and server SMTP will be compiled in. This is only necessary if your machine has some mailer that speaks SMTP.

DAEMON If set, code to run a daemon is compiled in. This code is for 4.2BSD if the NVMUNIX flag is specified; otherwise, 4.1a BSD code is used. Beware however that there are bugs in the 4.1a code that make it impossible for **sendmail** to work correctly under heavy load.

UGLYUUCP If you have a UUCP host adjacent to you which is not running a reasonable version of *rmail*, you will have to set this flag to include the "remote from sysname" info on the from line. Otherwise, UUCP gets confused about where the mail came from.

NOTUNIX If you are using a non-UNIX mail format, you can set this flag to turn off special processing of UNIX-style "From " lines.

**Configuration in src/conf.c**

Not all header semantics are defined in the configuration file. Header lines that should only be included by certain mailers (as well as other more obscure semantics) must be specified in the *HdrInfo* table in *conf.c*. This table contains the header name (which should be in all lower case) and a set of header control flags (described below). The flags are:

H_ACHECK Normally when the check is made to see if a header line is compatible with a mailer, *sendmail* will not delete an existing line. If this flag is set, *sendmail* will delete even existing header lines. That is, if this bit is set and the mailer does not have flag bits set that intersect with the required mailer flags in the header definition in sendmail.cf, the header line is *always* deleted.

H_EOH If this header field is set, treat it like a blank line, i.e., it will signal the end of the header and the beginning of the message text.

H_FORCE Add this header entry even if one existed in the message before. If a header entry does not have this bit set, *sendmail* will not add another header line if a header line of this name already existed. This would normally be used to stamp the message by everyone who handled it.

H_TRACE If set, this is a timestamp (trace) field. If the number of trace fields in a message exceeds a preset amount the message is returned on the assumption that it has an aliasing loop.

H_RCPT If set, this field contains recipient addresses. This is used by the –t flag to determine who to send to when it is collecting recipients from the message.

H_FROM This flag indicates that this field specifies a sender. The order of these fields in the *HdrInfo* table specifies *sendmail's* preference for which field to return error messages to.

Let's look at a sample *HdrInfo* specification:

```
struct hdrinfo          HdrInfo[] =
{
        /* originator fields, most to least significant */
    "resent-sender",    H_FROM,
    "resent-from",      H_FROM,
    "sender",           H_FROM,
    "from",             H_FROM,
    "full-name",        H_ACHECK,
        /* destination fields */
    "to",               H_RCPT,
    "resent-to",        H_RCPT,
    "cc",               H_RCPT,
        /* message identification and control */
    "message",          H_EOH,
    "text",             H_EOH,
        * trace fields */
    "received",         H_TRACE|H_FORCE,

    NULL,               0,
};
```

This structure indicates that the "To:", "Resent-To:", and "Cc:" fields all specify recipient addresses. Any "Full-Name:" field will be deleted unless the required mailer flag (indicated in the configuration file) is specified. The "Message:" and "Text:" fields will terminate the header; these are specified in new protocols [NBS80] or used by random dissenters around the network world. The "Received:" field will always be added, and can be used to trace messages.

There are a number of important points here. First, header fields are not added automatically just because they are in the *HdrInfo* structure; they must be specified in the configuration file in order to be added to the message. Any header fields mentioned in the configuration file but not mentioned in the *HdrInfo* structure have default processing performed; that is, they are added unless they were in the message already. Second, the *HdrInfo* structure only specifies cliched processing; certain headers are processed specially by ad hoc code regardless of the status specified in *HdrInfo*. For example, the "Sender:" and "From:" fields are always scanned on ARPANET mail to determine the sender; this is used to perform the "return to sender" function. The "From:" and "Full-Name:" fields are used to determine the full name of the sender if possible; this is stored in the macro **$x** and used in a number of ways.

The file *conf.c* also contains the specification of ARPANET reply codes. There are four classifications these fall into:

```
char Arpa_Info[] =      "050"; /* arbitrary info */
char Arpa_TSyserr[] =   "455"; /* some (transient) system error */
char Arpa_PSyserr[] =   "554"; /* some (transient) system error */
char Arpa_Usrerr[] =    "554"; /* some (fatal) user error */
```

The class *Arpa_Info* is for any information that is not required by the protocol, such as forwarding information. *Arpa_TSyserr* and *Arpa_PSyserr* is printed by the *syserr* routine. TSyserr is printed out for transient errors, whereas PSyserr is printed for permanent errors; the distinction is made based on the value of *errno*. Finally, *Arpa_Usrerr* is the result of a user error and is generated by the *usrerr* routine: these are generated when the user has specified something wrong, and hence the error is permanent, i.e., it will not work simply by resubmitting the request.

If it is necessary to restrict mail through a relay, the *checkcompat* routine can be modified. This routine is called for every recipient address. It can return **TRUE** to indicate that the address is acceptable and mail processing will continue, or it can return **FALSE** to reject the recipient. If it returns false, it is up to *checkcompat* to print an error message (using *usrerr*) saying why the message is rejected. For example, *checkcompat* could read:

```
bool
checkcompat(to)
    register ADDRESS *to;
{
    if (MsgSize > 50000 && to->q_mailer != LocalMailer)
    {
        usrerr("Message too large for non-local delivery");
        NoReturn = TRUE;
        return (FALSE);
    }
    return (TRUE);
}
```

This would reject messages greater than 50000 bytes unless they were local. The *NoReturn* flag can be sent to supress the return of the actual body of the message in the error return. The actual use of this routine is highly dependent on the implementation, and use should be limited.

# APPENDIX E

## SUMMARY OF SUPPORT FILES

This is a summary of the support files that *sendmail* creates or generates.

/usr/lib/sendmail
: The binary of *sendmail*.

/usr/bin/newaliases
: A link to /usr/lib/sendmail; causes the alias database to be rebuilt. Running this program is completely equivalent to giving *sendmail* the **-bi** flag.

/usr/bin/mailq
: Prints a listing of the mail queue. This program is equivalent to using the **-bp** flag to *sendmail*.

/usr/lib/sendmail.cf
: The configuration file, in textual form.

/usr/lib/sendmail.fc
: The configuration file represented as a memory image.

/usr/lib/sendmail.hf
: The SMTP help file.

/usr/lib/sendmail.st
: A statistics file; need not be present.

/usr/lib/aliases
: The textual version of the alias file.

/usr/lib/aliases.{pag,dir}
: The alias file in *dbm*(3) format.

/etc/syslog
: The program to do logging.

/etc/syslog.conf
: The configuration file for syslog.

/etc/syslog.pid
: Contains the process id of the currently running syslog.

/usr/spool/mqueue
: The directory in which the mail queue and temporary files reside.

/usr/spool/mqueue/qf*
: Control (queue) files for messages.

/usr/spool/mqueue/df*
: Data files.

/usr/spool/mqueue/lf*
: Lock files

/usr/spool/mqueue/tf*
: Temporary versions of the qf files, used during queue file rebuild.

/usr/spool/mqueue/nf*
: A file used when creating a unique id.

/usr/spool/mqueue/xf*
: A transcript of the current session.

Network Working Group
Request for Comments: 819

# Appendix G:
# The Domain Naming Convention for Internet User Applications

## 1. Introduction

For many years, the naming convention "<user>@<host>" has served the ARPANET user community for its mail system, and the substring "<host>" has been used for other applications such as file transfer (FTP) and terminal access (Telnet). With the advent of network interconnection, this naming convention needs to be generalized to accommodate internetworking. A decision has recently been reached to replace the simple name field, "<host>", by a composite name field, "<domain>" [2]. This note is an attempt to clarify this generalized naming convention, the Internet Naming Convention, and to explore the implications of its adoption for Internet name service and user applications.

The following example illustrates the changes in naming convention:

ARPANET Convention: Fred@ISIF
Internet Convention: Fred@F.ISI.ARPA

The intent is that the Internet names be used to form a tree-structured administrative dependent, rather than a strictly topology dependent, hierarchy. The left-to-right string of name components proceeds from the most specific to the most general, that is, the root of the tree, the administrative universe, is on the right.

The name service for realizing the Internet naming convention is assumed to be application independent. It is not a part of any particular application, but rather an independent name service serves different user applications.

## 2. The Structural Model

The Internet naming convention is based on the domain concept. The name of a domain consists of a concatenation of one or more <simple names>. A domain can be considered as a region of jurisdiction for name assignment and of responsibility for name-to-address translation. The set of domains forms a hierarchy.

Using a graph theory representation, this hierarchy may be modeled as a directed graph. A directed graph consists of a set of nodes and a

collection of arcs, where arcs are identified by ordered pairs of distinct nodes [1]. Each node of the graph represents a domain. An ordered pair (B, A), an arc from B to A, indicates that B is a subdomain of domain A, and B is a simple name unique within A. We will refer to B as a child of A, and A a parent of B. The directed graph that best describes the naming hierarchy is called an "in-tree", which is a rooted tree with all arcs directed towards the root (Figure 1). The root of the tree represents the naming universe, ancestor of all domains. Endpoints (or leaves) of the tree are the lowest-level domains.

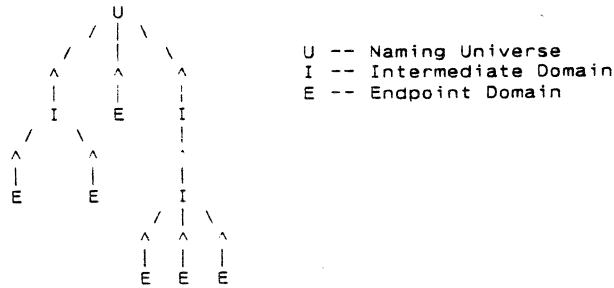



Figure 1
The In-Tree Model for Domain Hierarchy

The simple name of a child in this model is necessarily unique within its parent domain. Since the simple name of the child's parent is unique within the child's grandparent domain, the child can be uniquely named in its grandparent domain by the concatenation of its simple name followed by its parent's simple name.

For example, if the simple name of a child is "C1" then no other child of the same parent may be named "C1". Further, if the parent of this child is named "P1", then "P1" is a unique simple name in the child's grandparent domain. Thus, the concatenation C1.P1 is unique in C1's grandparent domain.

Similarly, each element of the hierarchy is uniquely named in the universe by its complete name, the concatenation of its simple name and those for the domains along the trail leading to the naming universe.

The hierarchical structure of the Internet naming convention supports decentralization of naming authority and distribution of name service capability. We assume a naming authority and a name server

associated with each domain. In Sections 5 and 6 respectively the name service and the naming authority are discussed.

Within an endpoint domain, unique names are assigned to <user> representing user mailboxes. User mailboxes may be viewed as children of their respective domains.

In reality, anomalies may exist violating the in-tree model of naming hierarchy. Overlapping domains imply multiple parentage, i.e., an entity of the naming hierarchy being a child of more than one domain. It is conceivable that ISI can be a member of the ARPA domain as well as a member of the USC domain (Figure 2). Such a relation constitutes an anomaly to the rule of one-connectivity between any two points of a tree. The common child and the sub-tree below it become descendants of both parent domains.

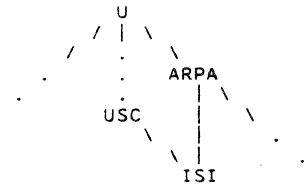



Figure 2
Anomaly in the In-Tree Model

Some issues resulting from multiple parentage are addressed in Appendix B. The general implications of multiple parentage are a subject for further investigation.

## 3. Advantage of Absolute Naming

Absolute naming implies that the (complete) names are assigned with respect to a universal reference point. The advantage of absolute naming is that a name thus assigned can be universally interpreted with respect to the universal reference point. The Internet naming convention provides absolute naming with the naming universe as its universal reference point.

For relative naming, an entity is named depending upon the position of the naming entity relative to that of the named entity. A set of hosts running the "unix" operating system exchange mail using a method called "uucp". The naming convention employed by uucp is an example of relative naming. The mail recipient is typically named by a source route identifying a chain of locally known hosts linking the

sender's host to the recipient's. A destination name can be, for example,

"alpha!beta!gamma!john",

where "alpha" is presumably known to the originating host, "beta" is known to "alpha", and so on.

The uucp mail system has demonstrated many of the problems inherent to relative naming. When the host names are only locally interpretable, routing optimization becomes impossible. A reply message may have to traverse the reverse route to the original sender in order to be forwarded to other parties.

Furthermore, if a message is forwarded by one of the original recipients or passed on as the text of another message, the frame of reference of the relative source route can be completely lost. Such relative naming schemes have severe problems for many of the uses that we depend upon in the ARPA Internet community.

## 4. Interoperability

To allow interoperation with a different naming convention, the names assigned by a foreign naming convention need to be accommodated. Given the autonomous nature of domains, a foreign naming environment may be incorporated as a domain anywhere in the hierarchy. Within the naming universe, the name service for a domain is provided within that domain. Thus, a foreign naming convention can be independent of the Internet naming convention. What is implied here is that no standard convention for naming needs to be imposed to allow interoperations among heterogeneous naming environments.

For example:

There might be a naming convention, say, in the FOO world, something like "<user>%<host>%<area>". Communications with an entity in that environment can be achieved from the Internet community by simply appending ".FOO" on the end of the name in that foreign convention.

John%ISI-Tops20-7%California.FOO

Another example:

One way of accommodating the "uucp world" described in the last section is to declare it as a foreign system. Thus, a uucp name

"alpha!beta!gamma!john"

might be known in the Internet community as

"alpha!beta!gamma!john.UUCP".

Communicating with a complex subdomain is another case which can be treated as interoperation. A complex subdomain is a domain with complex internal naming structure presumably unknown to the outside world (or the outside world does not care to be concerned with its complexity).

For the mail system application, the names embedded in the message text are often used by the destination for such purpose as to reply to the original message. Thus, the embedded names may need to be converted for the benefit of the name server in the destination environment.

Conversion of names on the boundary between heterogeneous naming environments is a complex subject. The following example illustrates some of the involved issues.

For example:

A message is sent from the Internet community to the FOO environment. It may bear the "From" and "To" fields as:

From: Fred@F.ISI.ARPA
To: John%ISI-Tops20-7%California.FOO

where "FOO" is a domain independent of the Internet naming environment. The interface on the boundary of the two environments may be represented by a software module. We may assume this interface to be an entity of the Internet community as well as an entity of the FOO community. For the benefit of the FOO environment, the "From" and "To" fields need to be modified upon the message's arrival at the boundary. One may view naming as a separate layer of protocol, and treat conversion as a protocol translation. The matter is complicated when the message is sent to more than one destination within different naming environments; or the message is destined within an environment not sharing boundary with the originating naming environment.

While the general subject concerning conversion is beyond the scope of this note, a few questions are raised in Appendix D.

## 5. Name Service

Name service is a network service providing name-to-address translation. Such service may be achieved in a number of ways. For a simple networking environment, it can be accomplished with a single central database containing name-to-address correspondence for all the pertinent network entities, such as hosts.

In the case of the old ARPANET host names, a central database is duplicated in each individual host. The originating module of an application process would query the local name service (e.g., make a system call) to obtain network address for the destination host. With the proliferation of networks and an accelerating increase in the number of hosts participating in networking, the ever growing size, update frequency, and the dissemination of the central database makes this approach unmanageable.

The hierarchical structure of the Internet naming convention supports decentralization of naming authority and distribution of name service capability. It readily accommodates growth of the naming universe. It allows an arbitrary number of hierarchical layers. The addition of a new domain adds little complexity to an existing Internet system.

The name service at each domain is assumed to be provided by one or more name servers. There are two models for how a name server completes its work, these might be called "iterative" and "recursive".

> For an iterative name server there may be two kinds of responses. The first kind of response is a destination address. The second kind of response is the address of another name server. If the response is a destination address, then the query is satisfied. If the response is the address of another name server, then the query must be repeated using that name server, and so on until a destination address is obtained.
>
> For a recursive name server there is only one kind of response -- a destination address. This puts an obligation on the name server to actually make the call on another name server if it can't answer the query itself.

It is noted that looping can be avoided since the names presented for translation can only be of finite concatenation. However, care should be taken in employing mechanisms such as a pointer to the next simple name for resolution.

We believe that some name servers will be recursive, but we don't believe that all will be. This means that the caller must be

prepared for either type of server. Further discussion and examples of name service is given in Appendix C.

The basic name service at each domain is the translation of simple names to addresses for all of its children. However, if only this basic name service is provided, the use of complete (or fully qualified) names would be required. Such requirement can be unreasonable in practice. Thus, we propose the use of partial names in the context in which their uniqueness is preserved. By construction, naming uniqueness is preserved within the domain of a common ancestry. Thus, a partially qualified name is constructed by omitting from the complete name ancestors common to the communicating parties. When a partially qualified name leaves its context of uniqueness it must be additionally qualified.

The use of partially qualified names places a requirement on the Internet name service. To satisfy this requirement, the name service at each domain must be capable of, in addition to the basic service, resolving simple names for all of its ancestors (including itself) and their children. In Appendix B, the required distinction among simple names for such resolution is addressed.

## 6. Naming Authority

Associated with each domain there must be a naming authority to assign simple names and ensure proper distinction among simple names.

Note that if the use of partially qualified names is allowed in a sub-domain, the uniqueness of simple names inside that sub-domain is insufficient to avoid ambiguity with names outside the subdomain. Appendix B discusses simple name assignment in a sub-domain that would allow the use of partially qualified names without ambiguity.

Administratively, associated with each domain there is a single person (or office) called the registrar. The registrar of the naming universe specifies the top-level set of domains and designates a registrar for each of these domains. The registrar for any given domain maintains the naming authority for that domain.

## 7. Network-Oriented Applications

For user applications such as file transfer and terminal access, the remote host needs to be named. To be compatible with ARPANET naming convention, a host can be treated as an endpoint domain.

Many operating systems or programming language run-time environments provide functions or calls (JSYSs, SVCs, UUOs, SYSs, etc.) for standard services (e.g., time-of-day, account-of-logged-in-user, convert-number-to-string). It is likely to be very helpful if such a

function or call is developed for translating a host name to an address. Indeed, several systems on the ARPANET already have such facilities for translating an ARPANET host name into an ARPANET address based on internal tables.

We recommend that this provision of a standard function or call for translating names to addresses be extended to accept names of Internet convention. This will promote a consistent interface to the users of programs involving internetwork activities. The standard facility for translating Internet names to Internet addresses should include all the mechanisms available on the host, such as checking a local table or cache of recently checked names, or consulting a name server via the Internet.

8. Mail Relaying

Relaying is a feature adopted by more and more mail systems. Relaying facilitates, among other things, interoperations between heterogeneous mail systems. The term "relay" is used to describe the situation where a message is routed via one or more intermediate points between the sender and the recipient. The mail relays are normally specified explicitly as relay points in the instructions for message delivery. Usually, each of the intermediate relays assume responsibility for the relayed message [3].

> A point should be made on the basic difference between mail relaying and the uucp naming system. The difference is that although mail relaying with absolute naming can also be considered as a form of source routing, the names of each intermediate points and that of the destination are universally interpretable, while the host names along a source route of the uucp convention is relative and thus only locally interpretable.

The Internet naming convention explicitly allows interoperations among heterogeneous systems. This implies that the originator of a communication may name a destination which resides in a foreign system. The probability is that the destination network address may not be comprehensible to the transport system of the originator. Thus, an implicit relaying mechanism is called for at the boundary between the domains. The function of this implicit relay is the same as the explicit relay.

## 9. Implementation

### The Actual Domains

The initial set of top-level names include:

ARPA

> This represents the set of organizations involved in the Internet system through the authority of the U.S. Defense Advanced Research Projects Agency. This includes all the research and development hosts on the ARPANET and hosts on many other nets as well. But note very carefully that the top-level domain "ARPA" does not map one-to-one with the ARPANET -- domains are administrative, not topological.

### Transition

In the transition from the ARPANET naming convention to the Internet naming convention, a host name may be used as a simple name for an endpoint domain. Thus, if "USC-ISIF" is an ARPANET host name, then "USC-ISIF.ARPA" is the name of an Internet domain.

## 10. Summary

A hierarchical naming convention based on the domain concept has been adopted by the Internet community. It is an absolute naming convention defined along administrative rather than topological boundaries. This naming convention is adaptive for interoperations with other naming conventions. Thus, no standard convention needs to be imposed for interoperations among heterogeneous naming environments.

This Internet naming convention allows distributed name service and naming authority functions at each domain. We have specified these functions required at each domain. Also discussed are implications on network-oriented applications, mail systems, and administrative aspects of this convention.

APPENDIX A

The BNF Specification

We present here a rather detailed "BNF" definition of the allowed form for a computer mail "mailbox" composed of a "local-part" and a "domain" (separated by an at sign). Clearly, the domain can be used separately in other network-oriented applications.

<mailbox> ::= <local-part> "@" <domain>

<local-part> ::= <string> | <quoted-string>

<string> ::= <char> | <char> <string>

<quoted-string> ::= """ <qtext> """

<qtext> ::= "\" <x> | "\" <x> <qtext> | <q> | <q> <qtext>

<char> ::= <c> | "\" <x>

<domain> ::= <naming-domain> | <naming-domain> "." <domain>

<naming-domain> ::= <simple-name> | <address>

<simple-name> ::= <a> <ldh-str> <let-dig>

<ldh-str> ::= <let-dig-hyp> | <let-dig-hyp> <ldh-str>

<let-dig> ::= <a> | <d>

<let-dig-hyp> ::= <a> | <d> | "-"

<address> :: = "#" <number> | "[" <dotnum> "]"

<number> ::= <d> | <d> <number>

<dotnum> ::= <snum> "." <snum> "." <snum> "." <snum>

<snum> ::= one, two, or three digits representing a decimal integer value in the range 0 through 255

<a> ::= any one of the 52 alphabetic characters A through Z in upper case and a through z in lower case

<c> ::= any one of the 128 ASCII characters except <s> or <SP>

<d> ::= any one of the ten digits 0 through 9

<q> ::= any one of the 128 ASCII characters except CR, LF, quote ("), or backslash (\)

<x> ::= any one of the 128 ASCII characters (no exceptions)

<s> ::= "<", ">", "(", ")", "[", "]", "\", ".", ",", ";", ":", "@", """, and the control characters (ASCII codes 0 through 31 inclusive and 127)

Note that the backslash, "\", is a quote character, which is used to indicate that the next character is to be used literally (instead of its normal interpretation). For example, "Joe\,Smith" could be used to indicate a single nine character user field with comma being the fourth character of the field.

The simple names that make up a domain may contain both upper and lower case letters (as well as digits and hyphen), but these names are not case sensitive.

Hosts are generally known by names. Sometimes a host is not known to the translation function and communication is blocked. To bypass this barrier two forms of addresses are also allowed for host "names". One form is a decimal integer prefixed by a pound sign, "#". Another form, called "dotted decimal", is four small decimal integers separated by dots and enclosed by brackets, e.g., "[123.255.37.2]", which indicates a 32-bit ARPA Internet Address in four 8-bit fields. (Of course, these numeric address forms are specific to the Internet, other forms may have to be provided if this problem arises in other transport systems.)

APPENDIX B

An Aside on the Assignment of Simple Names

In the following example, there are two naming hierarchies joining at the naming universe 'U'. One consists of domains (S, R, N, J, P, Q, B, A); and the other (L, E, F, G, H, D, C, K, B, A). Domain B is assumed to have multiple parentage as shown.

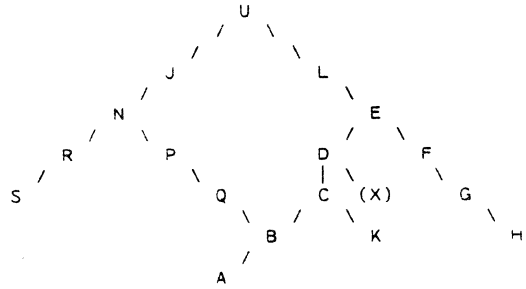



Figure 3
Illustration of Requirements for the Distinction of Simple Names

Suppose someone at A tries to initiate communication with destination H. The fully qualified destination name would be .

H.G.F.E.L.U

Omitting common ancestors, the partially qualified name for the destination would be

H.G.F

To permit the case of partially qualified names, name server at A needs to resolve the simple name F, i.e., F needs to be distinct from all the other simple names in its database.

To enable the name server of a domain to resolve simple names, a simple name for a child needs to be assigned distinct from those of all of its ancestors and their immediate children. However, such distinction would not be sufficient to allow simple name resolution at lower-level domains because lower-level domains could have multiple parentage below the level of this domain.

In the example above, let us assume that a name is to be assigned

to a new domain X by D. To allow name server at D to resolve simple names, the name for X must be distinct from L, E, D, C, F, and J. However, allowing A to resolve simple names, X needs to be also distinct from A, B, K, as well as from Q, P, N, and R.

The following observations can be made.

Simple names along parallel trails (distinct trails leading from one domain to the naming universe) must be distinct, e.g., N must be distinct from E for B or A to properly resolve simple names.

No universal uniqueness of simple names is called for, e.g., the simple name S does not have to be distinct from that of E, F, G, H, D, C, K, Q, B, or A.

The lower the level at which a domain occurs, the more immune it is to the requirement of naming uniqueness.

To satisfy the required distinction of simple names for proper resolution at all levels, a naming authority needs to ensure the simple name to be assigned distinct from those in the name server databases at the endpoint naming domains within its domain. As an example, for D to assign a simple name for X, it would need to consult databases at A and K. It is, however, acceptable to have simple names under domain A identical with those under K. Failure of such distinct assignment of simple names by naming authority of one domain would jeopardize the capability of simple name resolution for entities within the subtree under that domain.

APPENDIX C

Further Discussion of Name Service and Name Servers

The name service on a system should appear to the programmer of an application program simply as a system call or library subroutine. Within that call or subroutine there may be several types of methods for resolving the name string into an address.

First, a local table may be consulted. This table may be a complete table and may be updated frequently, or it may simply be a cache of the few latest name to address mappings recently determined.

Second, a call may be made to a name server to resolve the string into a destination address.

Third, a call may be made to a name server to resolve the string into a relay address.

Whenever a name server is called it may be a recursive server or an interactive server.

If the server is recursive, the caller won't be able to tell if the server itself had the information to resolve the query or called another server recursively (except perhaps for the time it takes).

If the server is iterative, the caller must be prepared for either the answer to its query, or a response indicating that it should call on a different server.

It should be noted that the main name service discussed in this memo is simply a name string to address service. For some applications there may be other services needed.

For example, even within the Internet there are several procedures or protocols for actually transferring mail. One need is to determine which mail procedures a destination host can use. Another need is to determine the name of a relay host if the source and destination hosts do not have a common mail procedure. These more specialized services must be specific to each application since the answers may be application dependent, but the basic name to address translation is application independent.

APPENDIX D

Further Discussion of Interoperability and Protocol Translations

The translation of protocols from one system to another is often quite difficult. Following are some questions that stem from considering the translations of addresses between mail systems:

What is the impact of different addressing environments (i.e., environments of different address formats)?

It is noted that the boundary of naming environment may or may not coincide with the boundary of different mail systems. Should the conversion of naming be independent of the application system?

The boundary between different addressing environments may or may not coincide with that of different naming environments or application systems. Some generic approach appears to be necessary.

If the conversion of naming is to be independent of the application system, some form of interaction appears necessary between the interface module of naming conversion with some application level functions, such as the parsing and modification of message text.

To accommodate encryption, conversion may not be desirable at all. What then can be an alternative to conversion?

GLOSSARY

address

An address is a numerical identifier for the topological location of the named entity.

name

A name is an alphanumeric identifier associated with the named entity. For unique identification, a name needs to be unique in the context in which the name is used. A name can be mapped to an address.

complete (fully qualified) name

A complete name is a concatenation of simple names representing the hierarchical relation of the named with respect to the naming universe, that is it is the concatenation of the simple names of the domain structure tree nodes starting with its own name and ending with the top level node name. It is a unique name in the naming universe.

partially qualified name

A partially qualified name is an abbreviation of the complete name omitting simple names of the common ancestors of the communicating parties.

simple name

A simple name is an alphanumeric identifier unique only within its parent domain.

domain

A domain defines a region of jurisdiction for name assignment and of responsibility for name-to-address translation.

naming universe

Naming universe is the ancestor of all network entities.

naming environment

A networking environment employing a specific naming convention.

name service

Name service is a network service for name-to-address mapping.

name server

A name server is a network mechanism (e.g., a process) realizing the function of name service.

naming authority

Naming authority is an administrative entity having the authority for assigning simple names and responsibility for resolving naming conflict.

parallel relations

A network entity may have one or more hierarchical relations with respect to the naming universe. Such multiple relations are parallel relations to each other.

multiple parentage

A network entity has multiple parentage when it is assigned a simple name by more than one naming domain.

REFERENCES

[1] F. Harary, "Graph Theory", Addison-Wesley, Reading, Massachusetts, 1969.

[2] J. Postel, "Computer Mail Meeting Notes", RFC-805, USC/Information Sciences Institute, 8 February 1982.

[3] J. Postel, "Simple Mail Transfer Protocol", RFC-821, USC/Information Sciences Institute, August 1982.

[4] D. Crocker, "Standard for the Format of ARPA Internet Text Messages", RFC-822, Department of Electrical Engineering, University of Delaware, August 1982.

**Silicon Graphics, Inc.**

Date _____

Your name _____

Title _____

Department _____

Company _____

Address _____

_____

_____

Phone _____

**COMMENTS**

Manual title and version _____

_____

Please list any errors, inaccuracies, or omissions you have found in this manual

_____

_____

_____

_____

Please list any suggestions you may have for improving this manual

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 45 MOUNTAIN VIEW, CA

POSTAGE WILL BE PAID BY ADDRESSEE

**Silicon Graphics, Inc.**
Attention: Technical Publications
2011 Stierlin Road
Mountain View, CA 94043-1321

# TCP/IP User's Guide

*Version 3.0*

Silicon Graphics, Inc.
2011 Stierlin Road
Mountain View, CA 94043

Document Number 007-0330-030

**Technical Publications:**

Gail Kesner
Diane Wilford

**Engineering:**

Vernon Schryver
Archer Sully



**TCP/IP User's Guide**
Document Number 007-0330-030

CONTENTS

# 1. Introduction

This book is designed for people who want to use the IRIS Series 3000 workstation to communicate with other computers using TCP/IP. In this book, IRIS Series 3000 indicates all models in the 3000 and 3100 product lines, including the series 2000 Turbo products (2400T and 2500T).

If this is your first time using an IRIS workstation, *Getting Started with Your IRIS Workstation* is the first book you should read. To use the procedures in this book, your IRIS must be running with the Ethernet card installed. If your IRIS is not running, see Chapter 2, Hardware Installation and Chapter 3, Booting the IRIS, in the *IRIS Series 3000 Owner's Guide.* If the Ethernet card is not installed, call the Field Engineer who services your account.

This document explains how to configure and use Transmission Control Protocol/Internet Protocol (TCP/IP) communications software on the IRIS Series 3000 workstation.

The IRIS can communicate across an Ethernet local area network with other hosts and terminals using TCP/IP communications software. TCP/IP offers file transfer and remote login services. To connect the IRIS to the Ethernet, see Chapter 2, Hardware Installation, in the *IRIS Series 3000 Owner's Guide.*

The IRIS workstation runs the 4.3 BSD UNIX version of TCP/IP except that the IRIS does not support the program *talk* and the UNIX domain socket, UNIX_AF.

## 1.1 Getting Started

This document makes some assumptions about you and your IRIS 3000 Series workstation:

- Your system is up and running and you know how to boot it. If not, consult the *IRIS Series 3000 Owner's Guide.*
- You know a little about the UNIX operating system. If not, read *Getting Started with Your IRIS Workstation.*
- You are familiar with a text editor. If not, read *Getting Started with Your IRIS Workstation.*

## 1.2 TCP/IP Overview

TCP/IP is the basic network protocol supported on IRIS Series 3000 workstations. A communications protocol is a procedure with a well defined format that allows two or more systems to communicate across a physical link. On the IRIS, the physical link is the Ethernet network.

TCP/IP provides a fast, standard, and reliable means of communicating with other systems running UNIX on your network.

TCP/IP commands let you:

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

Below is a diagram of a typical Ethernet network.

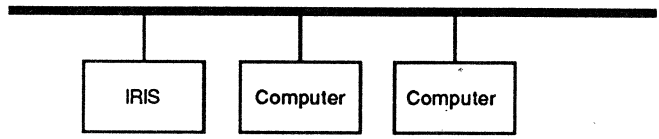



Figure 1-1: Ethernet Network

TCP/IP software is a standard feature on the IRIS. To use TCP/IP, you must have the appropriate hardware:

- an Ethernet cable
- a drop cable from the IRIS to a transceiver
- an Ethernet board

Each computer on the network is linked to the Ethernet cable by a drop cable. The Ethernet board is available as an option with the IRIS Series 3000 workstation.

The Ethernet cable is the physical foundation of the network; it supports several layers of software. The first layer, Internet Protocol (IP), supports the Transmission Control Protocol (TCP).

Figure 1-2 shows a diagram of these network layers.

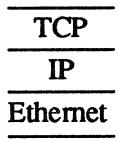



Figure 1-2: Layers of TCP/IP Network Software

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, and 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. The TCP layer enables and supports applications such as TCP/IP commands.

## 1.3 Conventions

This document uses the standard UNIX convention when referring to entries in the UNIX documentation. The entry name is followed by a section number in parentheses. For example, *cc*(1) refers to the *cc* manual entry in Section 1 of the *UNIX Programmer's Manual, Volume IA.*

In command syntax descriptions and examples, square brackets surrounding an argument indicate that the argument is optional. Variable parameters are in *italics*. You replace these variable with the appropriate string or value.

In text descriptions, filenames and UNIX commands are also in *italics*. IRIS Graphics Library routines and PROM commands are in `tpewriter font`.

In examples that are set off from the text, text that the machine displays is in `typewriter font`; text that you type is in **`bold typewriter font`**.

## 1.4 Relevant Documentation

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

- *Defense Data Network Protocol Handbook,* which is available from the Network Information Center, Defense Data Network, SRI International, Room EJ-291, 333 Ravenswood Ave., Menlo Park, California 94025, telephone: (415) 326-6200
- *IRIS Series 3000 Owner's Guide*
- *UNIX Programmer's Manual, Volume IA*
- *UNIX Programmer's Manual, Volume IB*

This table lists relevant manual pages from the *UNIX Programmer's Manual, Volume IA* and *UNIX Programmer's Manual, Volume IB.*

| Manual Page | Explanation |
|---|---|
| arp(1M) | Address resolution display and control |
| ftp(1C) | File transfer program |
| ftpd(1M) | DARPA Internet File Transfer Protocol server |
| ifconfig(1M) | Configure network interface parameters |
| inetd(1M) | Internet "super-server" |
| netstat(1) | Show network status |
| rcp(1C) | Copy file to or from a remote system |
| rlogin(1C) | Login to a remote system |
| rlogind(1M) | Remote login server |
| route(1M) | Manually manipulate the routing tables |
| routed(1M) | Network routing daemon |
| rsh(1C) | Start a remote shell |
| rshd(1M) | Remote shell server |
| ruptime(1C) | Show status of other hosts on local network |

Table 1-1: Manual Pages Relevant to TCP/IP

| Manual Page | Explanation |
|---|---|
| rwho(1C) | Show who is logged in on local machines |
| rwhod(1M) | System status server |
| telnet(1C) | User interface to TELNET Protocol |
| telnetd(1M) | TELNET Protocol server |
| accept(2) | Accept a connection on a socket |
| bind(2) | Bind a name to a socket |
| connect(2) | Initiate a connection on a socket |
| select(2) | Select I/O descriptor sets |
| send(2) | Send message to a socket |
| socket(2) | Create an endpoint for communication |
| byteorder(3N) | Convert values between host and network byte order |
| inet(3N) | Internet Protocol Family |
| rcmd(3N) | Return a stream |
| hosts(4) | Host name data base |
| hosts.equiv(4) | Host names for shared accounts |
| networks(4) | network name data base |
| protocols(4) | Protocol name data base |
| rhosts(4) | Host and user names in shared accounts |
| services(4) | Service name data base |
| ip(7P) | Internet Protocol |
| tcp(7P) | Internet Transmission Control Protocol |

Table 1-1: Manual Pages Relevant to TCP/IP (continued)

## 1.5 Product Support

Silicon Graphics, Inc., provides a comprehensive product support and maintenance program for IRIS products. For further information, contact Product Support through the Geometry Hotline

| Silicon Graphics Geometry Hotline | |
|---|---|
| (800) 345-0222 | U.S. and Canada (toll-free) |
| 350613 | Worldwide (telex number) |

# 2. Software Administration

This chapter describes the procedures for configuring TCP/IP communications software. Read this chapter if you are acting as either the system or network administrator for your IRIS Series 3000 workstation. If your system is already configured, go to Chapter 3. If your system is not connected to the Ethernet, see Chapter 2, Hardware Installation, in the *IRIS Series 3000 Owner's Guide*. This chapter includes these basic configuration procedures:

| Procedures | Section |
|---|---|
| Choosing an address class | 2.1 |
| Naming your IRIS workstation | 2.3 |
| Configuring TCP/IP software | 2.4 |
| Connecting networks | 2.6 |
| Setting up network mail | 2.7.1 |

The following checklist provides instructions on the necessary steps for configuring the TCP/IP software. Each step is explained in detail in this manual. You must follow the steps listed below for all machines.

- Choose an address class and assign an address for your system (Section 2.1). If you are connecting your system to an existing network, ask your network administrator for the address you should assign to your system.
- Select your host number (Section 2.2).
- Assign a unique name to your system (Section 2.3). The default name is *IRIS*.
- Create the host database for your system by editing the file */etc/hosts*. Edit the file */etc/hosts.equiv* for controlling host access (Section 2.4).
- Update the databases on other systems on the network with your new internet address and system name. For systems with the UNIX operating system, enter the changes in the */etc/hosts* file (Section 2.4).
- If you have PROMs that are version 3.0.9 or later, set your network address into PROM memory (Section 2.8).
- Read about and try the network application programs described in Chapters 3 and 4. Among other functions, these programs send and receive files, and log in to remote systems.

Follow these steps if you plan to use your IRIS 3000 Series workstation for network mail or as a gateway.

- Create gateway mappings for routing table initialization by adding *route* entries to the file */etc/rc.tcp* (Section 2.6). If you are connecting your IRIS Series 3000 workstation to an existing network in which gateways have already been established, obtain the gateway addresses from your network administrator.
- If you want to use network mail, edit */etc/hosts* and */usr/lib/sendmail.cf* to add the names of remote hosts (Section 2.7).

## 2.1 Choosing an Address Class

TCP/IP uses two addresses to uniquely identify each node in a network: a physical Ethernet address and a logical internet address. The Ethernet address is a 6-byte physical address that identifies the Ethernet board. Because it is burned into the hardware, it never changes.

The internet address is a 4-byte logical address that identifies the host on the network. The logical address is in two parts: the network number followed by the host number. You can specify addresses in decimal, octal, or hexadecimal. Use a leading 0x or 0X for hexadecimal notation, and use a leading 0 for octal notation. See *inet*(3N) for more information on the dot (.) notation.

There are three internet address classes: Class A, Class B, and Class C. The most significant bits of the address are the first byte that determine the address class. Each address has the form:

| net | host |
|---|---|

All internet addresses on a particular network must be of the same class.

The number of bytes comprising the `net` and `host` portions of address differs according to the address class. These differences are illustrated in Figure 2-1.

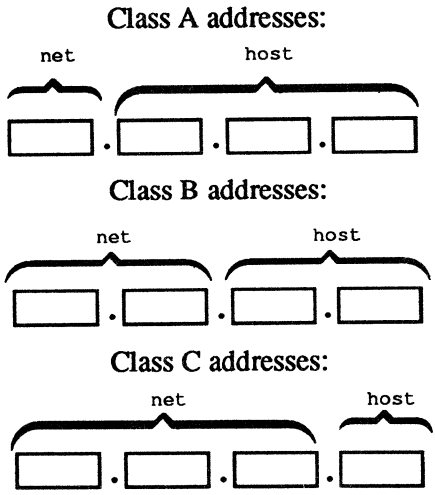



Figure 2-1: Net and Host Divisions of Addresses

To specify an internet address, you must list all four bytes of the address. If any of the bytes is zero (0), you must use a 0 to specify that byte (see below for examples).

**NOTE:** If you are connecting your system to an existing network, ask your network administrator for the appropriate network address.

In any network, all internet addresses must be of the same class. If you are connecting your system to an existing network, ask your network administrator for the address class and network number. If you are building a new network, see the information on assigning network classes below.

The IRIS Series 3000 workstation is shipped with the Class C address:

```
192.0.0.1
```

This address is in the */etc/hosts* file. If you do not plan to connect your network to a larger network and you do not already have a network number, 192.0.0.1 is a generic network number that you can use.

If you plan to connect your local network to a larger network, you must apply for an address through the agency that administers the network to which you wish to connect. For example, the Network Information Center

(NIC) at Stanford Research Institute (SRI) is responsible for all network numbers assignments for the Internet. If you have questions about Internet network numbers or want further network information, call 1-800-235-3155 or contact the Hostmaster at the NIC. The network address is:

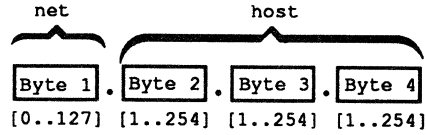
```
HOSTMASTER@SRI-NIC.ARPA
```

Each of the three classes of network is described below.

### 2.1.1 Class A Internet Addresses

As Figure 2-1 illustrates, in a class A address, the `net` portion of the address consists of one byte, and the `host` portion consists of three bytes.

In a class A address, the most significant bit of the `net` portion of the address must be 0. Because the most significant bit is 0, the possible range for the network identification number is from 1 to 127 (decimal), instead of 1 to 254. The `host` portion of the address consists of three bytes (24 bits). You must specify each byte of the 24-bit host address separately using dot notation. The possible range of values for each byte of the host address is 1 to 254. The possible range of values (in decimal) for each byte of the address is shown below:



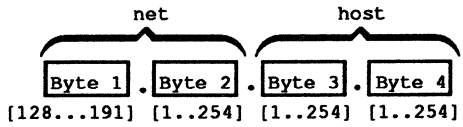

This is an example of a class A address:

```
42.0.0.1
```

In the above address, the network identification number is 42. It uses one byte of the specification. The host address is 1. Notice that it only uses one byte of three possible bytes. The other two bytes are 0.

## 2.1.2 Class B Internet Addresses

As Figure 2-1 illustrates, in a Class B address the `net` and `host` portions each consist of two bytes. In a Class B address, the two most significant bits of the first byte of the `net` portion of the address must be 10 (in binary). If the most significant bits are 10, then you must use only the remaining six bits of the first byte and all eight bits of the second byte to specify the rest of the network identification number. This means the possible range (in decimal) for the first byte of the `net` portion of the address is 128 to 191 and the range for the second byte is 1 to 254. Because the `host` portion of the address also consists of two bytes, there are $2^{16}$ possible host identification numbers. The possible range (in decimal) for the two bytes of the `host` portion of the address is 1 to 254. The possible range of values (in decimal) for each byte is shown below:




This is an example of a class B address:

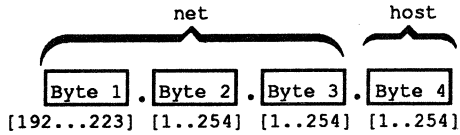
```
128.0.0.1
```

In the above address, the network identification number is 128.0. It uses two bytes of the specification. The host number is 1. The host address uses only one byte in the 2-byte specification. The other byte is specified with a 0.

## 2.1.3 Class C Internet Addresses

As shown in Figure 2-1, in a Class C address the `net` portion of the address consists of three bytes and the `host` portion of the address consists of one byte. In a Class C address, the three most significant bits of the `net` portion of the address must be 110 (in binary). If the most significant bits are 110, then you use the remaining five bits of the first byte and the 16 bits

of the second and third bytes to specify the rest of the network identification number. This means there are $2^{21}$ network identification numbers. For the `net` portion of the address, the possible range of values (in decimal) is 192 to 223 for the first byte of the address and 1 to 254 for the second and third bytes. Since the `host` portion of the address consists of one byte, there are $2^8$ host identification numbers. This means the possible range of values for the host identification number is 1 to 254. The possible range of values (in decimal) for each byte is shown below:




This is an example of a class C address:

```
192.0.0.1
```

In the above address, the network identification number is 192.0.0. It uses three bytes of the specification. The host number is 1.

## 2.2 Selecting a Host Number

After you select an address class, you need to select a specific internet host number for your system. As previously mentioned, the TCP/IP protocols use logical internet addresses while the Ethernet hardware uses physical Ethernet addresses. Therefore, in order to send messages between two systems connected by the Ethernet, the systems must be able to translate the logical internet addresses into the physical Ethernet addresses. This translation is done with the Address Resolution Protocol (ARP).

The sending system broadcasts a message containing the destination system's internet address. The sending system then waits for the destination system to send back its physical Ethernet address. After the sending system receives the Ethernet address, it associates this Ethernet address with the destination system's internet address. The sending system then stores this translation and the Ethernet address in a table for later use. Each system on the network maintains a table of recently used address translations.

Thus, the ARP protocol allows you to choose any host number as long as it is unique and is the same class as the other systems on the network. The only number that you cannot use is 127. This is the local host entry number, which is used for testing. */etc/hosts* lists all system names and addresses that your system can access. See Section 2.4.1 for more information on the */etc/hosts* file.

Normally, hosts using the ARP method of address translation can communicate only with other hosts that support the ARP method. The IRIS TCP/IP software, however, supports communication with hosts that do not support ARP. To establish communication with a host that does not support ARP, add the mapping for its Ethernet address to the internet-to-Ethernet address translation table on the IRIS. To do this, use the *arp*(1M) utility.

To use the *arp* utility, follow these steps:

1. On the system that supports ARP, log in as *root.*

```
login: root
```

2. Add information about the system that does not support ARP (host-name) to the translation table. Type:

**arp -s** *host_name Ethernet_address* **pub**

The **-s** option tells *arp* to add the entry that follows to the translation table. The **pub** option publishes the entry. That is, it enables a host to respond to an ARP broadcast even if the specified internet address is not its own address. (See *arp*(1M) for more information.)

Once you add an entry to the table, it remains in the translation table until you reboot. If you want to make the entry you add permanent, follow these directions:

- Edit */etc/rc.tcp*. Find this line:

```
#  Add routing commands here
```

Add this line:

```
arp -s host_name Ethernet_address pub
```

3. To insert multiple entries in the translation table by reading them from a text file, type:

```
arp -f filename
```

4. To display all the current entries in the table, type:

```
arp -a
```

The system responds with a list, similar to this:

```
opus      (192.60.0.1)    at  8:0:14:60:0:1
george    (192.60.1.81)   at  8:0:14:60:0:81
fred      (192.60.0.65)   at  8:0:14:0:0:65
```

## 2.3 Naming Your IRIS Workstation

The default name of a new IRIS Series 3000 workstation is *IRIS*. If you have more than one system on a network, you must assign each system a unique name. The name can be up to eight characters long and cannot contain blank spaces.

The name must be in lowercase letters. As a general practice, do not assign users or hosts any name that contains uppercase letters. If you must have uppercase letters, then create a lowercase alias in */usr/lib/aliases*. Many systems are case-insensitive, and some convert uppercase letters to lower case. To change the name of a system, edit the files */etc/hosts* and */etc/sys_id* and reboot the system. See the manual pages for *hosts*(4) and */etc/sys_id*(4).

All the systems on your network must add the name of your workstation to their */etc/hosts* files. If you are adding a new machine to an existing network, edit the */etc/hosts* file on your new workstation to include the name of another system on your network. Then, log in to the other system and

copy the */etc/hosts* file from the old system to the new one. The network administrator should add the new system name to their */etc/hosts* file and update the */etc/hosts* file on the other systems.

## 2.4 Configuring TCP/IP Software

After you establish an address for your network and your system, you need to configure your TCP/IP software to fit your needs. Specifically, you must perform the following tasks:

- Add your new host name to the host data base
- Set remote access privileges

### 2.4.1 Adding a New Host

The file */etc/hosts* is the host names database. It contains mappings between the internet addresses and the names and aliases for the systems on the network. This file exists on each system on a network. When you reference a host by name in an application program, the application program uses this file to determine the internet address of the host to which you are referring.

The */etc/hosts* file can contain two elements:

- lines of text with two or more fields
- optional comments that begin with a pound sign (#) and continue to the end of the line

The first field of the line is the internet address, which consists of a network number and a host number. In the entry for your machine, the internet address contains the network number that you obtained or selected according to the instructions in Section 2.1.

**NOTE:** The first field must be followed by one or more spaces or tabs.

The second field contains the name of the system that is associated with the internet address in the first field. Figure 2-2 shows an example of an */etc/hosts* file.

```
#
# internet Host Database
# test entry
127.0.0.1 localhost
# my machine
31.0.3.13  IRIS
# active entries
31.0.0.48  cyrano
31.0.0.57  percival
31.0.0.13  arthur
31.0.3.59  zurich
31.0.0.66  opus
31.0.0.81  dagwood
31.0.0.10  mac
31.0.0.11  george
31.0.0.26  blondie
```

Figure 2-2: Example of an */etc/hosts* File

If you are connecting to an existing network, obtain a copy of the */etc/hosts* file from your network administrator. (Before using this file, make sure it conforms to the syntax specifications outlined in Section 2.1.) If you are building a new network, add the name and address for your system as well as for any other systems on the network to */etc/hosts*. If you are going to use your IRIS as a gateway, it requires two host names. You must put both names for your workstation in the */etc/hosts*. See Section 2.6.3.

Figure 2-2 shows an entry called *localhost*. This entry is a special address that you use to test the TCP/IP software. When you reference this address, the message is looped back internally; it is never physically transmitted across the network. This localhost entry is a Class A address with network number 127 and host number 1.

## 2.4.2 Setting Remote Access Privileges

The */etc/hosts.equiv* file contains a list of remote systems with which the local system is equivalent, that is, with which it shares account names. This file is used by the application programs *rlogin*, *rsh*, and *rcp* to allow remote access to accounts on the local system.

There are two different ways to set up remote access privileges:

- Create a list of systems that can access your system. To do this, you edit */etc/hosts.equiv*.
- Create a list of other users that can access your system. To do this, you edit *.rhosts*.

You can use one or both of these methods. Each line in the files */etc/hosts.equiv* and *.rhosts* lists the unique name for each remote host whose users can access the local machine.

*/etc/hosts.equiv* contains lines of text with one or more fields that are separated by exactly one blank space. A line consisting of a simple host name means that anyone may log in from that host.

Figure 2-3 shows an example */etc/hosts.equiv* file that corresponds to the */etc/hosts* file shown in Figure 2-2.

```
localhost
iris
cyrano
percival
arthur
zurich
opus
dagwood
mac
george
blondie
```

Figure 2-3: Example of an */etc/hosts.equiv* File

Each line in */etc/hosts.equiv* is the name of a host that can access the local system. If you are connecting your system to an existing network, ask your network or system administrator for the host names that you should include in this file. If you are building a new network, edit the */etc/hosts.equiv* so that it contains the names of hosts that should have access to the files on your system. Include only those hosts in *hosts.equiv* that are equivalent to yours in security and administration.

### 2.4.3 Customizing *.rhosts*

You can use the *.rhosts* file to control remote access to systems. This file serves the same purpose as */etc/hosts.equiv*, but gives access to individual users. The *.rhosts* file is located in a user's home directory and specifies the remote systems and users that can access the local system under the login name of that user's directory.

If you are listed in another user's *.rhosts* file on a remote machine, you can log in to that machine with your login id and you have all the same privileges as the user who listed you in their *.rhosts* file. For example, user *sam*'s *.rhosts* file is shown in Figure 2-4. In this example, *donna, fred,* and *rich* use the machine *iris* and have all of *sam*'s privileges on his local machine. The *.rhosts* file is used to validate a user when the name of the remote system does not exist in the */etc/hosts.equiv* file.

*.rhosts* contains lines of text with fields for host and user separated by exactly one space. Tab characters are not allowed. Each line contains the name of the remote systems and the users on that remote system who can access the local system.

```
iris fred
iris rich
iris donna
zurich edwardo
opus bloom
opus pickles
dagwood daisy
mac henry
blondie marilyn
```

Figure 2-4: Example of an *.rhosts* File

## 2.5 Network Security

This section contains information on two network security issues: network access control and the TCP/IP security algorithm.

### 2.5.1 Controlling Network Access

The configuration files described in Sections 2.4.2 and 2.4.3, */etc/hosts.equiv* and *.rhosts*, maintain network security by controlling user access from remote systems to the local system.

Users on remote systems whose systems are listed in the local system's */etc/hosts.equiv* file can gain access to all files on the local system without using a password. *hosts.equiv* makes users on one machine equivalent to users on another machine. For example, user *paul* is a local user on system *chess*. If system *chess* is listed in system *spider*'s *host.equiv* file, then *paul* can use the *rsh*, *rlogin*, and *rcp* commands without supplying a password. The *host.equiv* makes the remote user equivalent to the local user with the same user name. Therefore, *paul* is *paul* on both *chess* and *spider*. See *hosts.equiv*(4).

*/.rhosts* is in *root*'s home directory. This file allows the superuser or permitted users on a remote host to log in as the superuser on the local host without specifying a password. *hosts.equiv* does not permit *root* access. See *rhosts*(4).

Another network security feature is configuration file ownership. You must be logged in as *root* in order to edit most of the configuration files.

### Warnings

Use */.rhosts* only if all systems and their consoles are physically secure against unauthorized use. Given access to a console with */.rhosts* privileges, anyone 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.

Be very selective about the systems you add to *hosts.equiv* because adding a system to *hosts.equiv* makes all users on the remote system equivalent to users on the local system. Making users on systems equivalent can be dangerous. For example, if *host_A* lists *host_B* in its */etc/hosts.equiv* file, then a user with superuser privileges on *host_B* can create an account with any name that matches a user on *host_A*, thus gaining access to that user's files on *host_A*, if *host.equiv* is not restricted to specific users.

## 2.5.2 Securing Your Network

You can maintain network security and allow users to have access to several machines by using three strategies:

- When more than one machine has the same protection domain, all the machines within that protection domain have identical *hosts.equiv* files. All users, except for *root* in this case, can use the *rlogin*, *rcp*, and *rsh* commands without supplying a password.
- When the user *root* on all machines within the same protection domain is put in each machine's */.rhosts* file, any user logged in as *root* on one machine can use the *rlogin*, *rcp*, and *rsh* commands without supplying a password.

- When one user has the same name on several machines, this user is listed in the *.rhosts* file in his home directory on each machine. This user can use the *rlogin*, *rcp*, and *rsh* commands without supplying a password.

## 2.6 Connecting Networks

After you set up your local network, you can connect it to other networks. These networks must be connected physically through a *gateway*. A gateway is an intermediate system that controls the flow of information between two (or more) systems. A *route* is a sequence of host names through which information is sent to its destination host. A route may go through one or more gateways. When you send information to a host on a remote network, it looks like the information is sent directly to the host; however, the information is actually sent to a device that channels it to a remote host.

Some gateways are systems dedicated only to this function; others are hosts that perform gateway functions in addition to their regular functions. Figure 2-5 shows a diagram of two networks with a gateway.

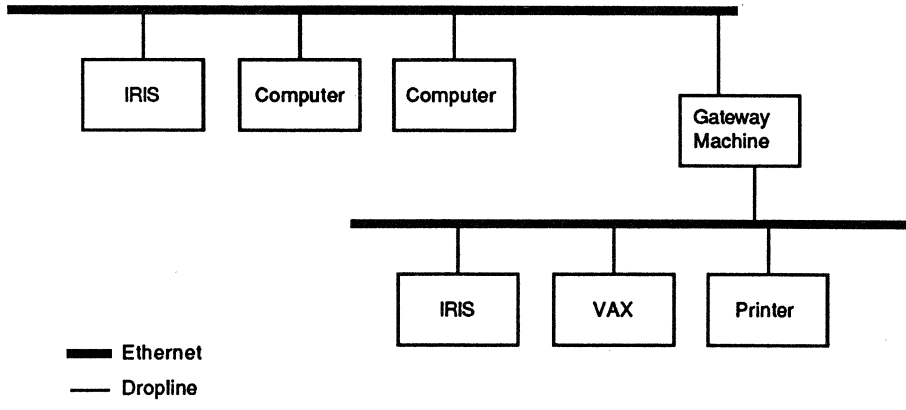



Figure 2-5: Example of Two Networks Connected by a Gateway

### 2.6.1 Linking to Another Network

To link to another network, you must include details about the network in the file */etc/networks*, which is the network host database. It contains mappings between networks and their internet addresses. You must be the superuser to edit this file. See *networks*(4).

Large networks, particularly those connected to the Internet, are administered by the Network Information Center (NIC), operated by Stanford Research Institute (SRI), in Palo Alto, CA. Contact the NIC for information on becoming a part of a large network.

You must create a map of the gateways that your system will use. You can create these gateways by using the *route* utility (Section 2.6.2) or by editing */etc/rc.tcp* (Section 2.6.3). All mappings between hosts and gateways

appear in the routing table. To access or change the mappings in the routing table use the *route*(1M) utility (Section 2.6.2).

When the network is up and running, the routing table resides in the TCP kernel. The initial entries in the table are based on the *route* commands in the */etc/rc.tcp* file.

Use the *netstat* –r command, described in Section 2.7.2, to display gateway entries in the routing table. Usually, the route daemon *routed* maintains the routing tables automatically.

### 2.6.2 *route* Utility

Many gateways run a version of the BSD networking software that supports the route daemon, *routed.* If you are connected to one of these networks, you do not need to read this section, because the gateway machine's *routed* is responsible for adding and deleting routes. Follow the directions in this section only if you are connecting to a network that does not use *routed* in its networking software.

The *route*(1M) utility lets you add or delete gateway addresses for internetwork communication. To add an address, type:

```
route add destination gateway metric
```

To delete an address, type:

```
route delete destination gateway metric
```

*destination* is either the name of a host on the remote network or it is the entire remote network. *gateway* is the gateway that connects the two networks. *metric* refers to the number of gateways information must go through to get to its destination. If you do not specify the metric value, *route* assumes a value of 0, which implies no gateways.

You can specify both *destination* and *gateway* as either names or internet addresses. Internet addresses must be in the same class as the network class. (For information on address classes, see Section 2.1.) You must list both *destination* and *gateway* in your */etc/hosts* file; *gateway* must be listed in */etc/networks*.

Below are examples of commands for adding and deleting a new route using the destination network. In each example, the system's response to the command contains the internet addresses (in four-part dot notation) that corresponds to the destination and gateway given in the command. See *inet*(7P) for information on dot notation.

**NOTE:** You must be logged in as *root* to add or delete gateways from the routing table.

## Adding a New Route

The following example adds a new route and gateway. The new route goes to the network *sanjose*. This route is through the gateway *sj*. The metric number indicates the number of gateways in this route. The metric number is one (1) in this example. For example, to add the route *san jose*, type:

```
su
route  add  sanjose  sj  1
```

After you enter this command, the system responds:

```
add  network  21.0.0.0  gateway  90.1.32.131
```

## Deleting a Route

The following example deletes a route and a gateway. To remove *sj* as a route through which the network *sanjose* can be reached, delete the entry from the routing table. For example, to delete the route *san jose*, type:

```
su
route  delete  sanjose  sj  1
```

After you enter this command, the system responds:

```
delete network 21.0.0.0 gateway 90.1.32.131
```

## Displaying Routes

To display a route, type:

```
netstat -r
```

The system displays this information:

```
Routing tables

Destination   Gateway          Flags   Refcnt   Use      Interface
localhost     localhost        UH         1     0        lo0
44            norman           UG         5     42151    ex0
194.26.51     alfred           U          1     18231    ex0
194.26.52     gate-TBsrc       U          1     65534    ex1
194.26.53     gate-owls        UG         5     3853     ex0
194.26.54     gate-goose       UG         2     28145    ex0
194.26.55     194.26.51.7      UG         1     741      ex0
```

Section 2.7.2 describes the *netstat* –r command in more detail.

## 2.6.3 Setting Up a Gateway

You must have two Ethernet boards in your IRIS Series 3000 workstation to use it as a gateway. Each Ethernet board has a unique Ethernet and internet address and a unique host name. The entry in */etc/hosts* for an IRIS that is a gateway would look like this:

```
191.25.50.10    gate-marketing
191.25.51.10    penquin
```

To use your IRIS Series 3000 workstation as a gateway, follow these steps:

1. Become the superuser. Type:

```
su
```

2. Use the file */etc/rc.tcp* to configure gateways, modify the routing table, and activate the *rwho* utility, if needed. *rc.tcp* is an initialization script that is invoked by */etc/init*, which is executed when you put the workstation into multi-user mode. *rc.tcp* performs TCP/IP-specific initializations on gateways. */usr/etc/ifconfig* initializes the Ethernet interface and causes *rc.tcp* to start the TCP/IP daemon *inetd*.

   Edit *rc.tcp* to make your workstation function as a gateway. Find these lines:

```
# change and install the following line
# for gatewaying
#    /etc/ifconfig ex1 inet gate-$HNAME

     /etc/ifconfig lo0 localhost

     hostid $HNAME
```

   You must give the workstation an alternative host name when you make it a gateway. You must put both of these names for your workstation in */etc/hosts*. To use the IRIS as a gateway, delete the pound sign (#) on the line with `gate-$HNAME`. This line causes your machine's host name to be defined as *gateway-hostname*. Below is an example of these lines in */etc/rc.tcp* changed to be used as a gateway.

```
# change and install the following line
# for gatewaying
     /etc/ifconfig ex1 inet gate-$HNAME

     /etc/ifconfig lo0 localhost

     hostid $HNAME
```

3. Edit */rc.tcp* to modify the script if you need to change the automatically inferred routing tables. Find this line and substitute either the host name or the internet address for either the destination or gateway machine.

```
# Example: "route add destination gateway 1"
```

Here is an example of the above line with the destination and gateway machines specified.

```
route add walrus northpole 2
```

You must use static routing to work with hosts that do not run *routed*(1M) or some other implementation of the RIP routing protocol. The easiest way to specify static routing is in */etc/gateways*, which is read by the local *routed*. To use static routing with *route*, follow these steps:

A. Log in as *root*.

```
login: root
```

B. Edit the file */etc/rc.tcp*. Search for the following comment:

```
# Add routing commands here
```

Below this comment is another comment that shows how to specify the gateway mappings:

```
# Example: "/usr/etc/net/route add destination
  gateway 1"
```

Add the gateway mappings for your system to the file. If you are connecting to a preexisting network, get the gateway addresses from your network administrator. For example, to map the gateway *sj* to the host *sanjose*, type:

```
/usr/etc/net/route add sanjose sj 2
```

C. To effect the changes you specify in gateway mapping, reboot the system.

## 2.7 Maintaining Your Network

This section provides information on network mail, statistics, and activating the *rwho* utility.

### 2.7.1 Setting up Network Mail

TCP/IP uses the *sendmail* utility. *sendmail* forwards internetwork using any network that will deliver the message to the correct place.

Each host running TCP/IP's network mail facility uses a configuration file, *sendmail.cf*, to store the names and communication modes of host machines. */usr/lib/aliases* stores user and network addresses. When you add a new machine to the network, you must edit */usr/lib/aliases* to update the list of users and their addresses to make these changes take effect, and then run the program *newaliases*.

#### The *sendmail.cf* File

The *sendmail.cf* file identifies and contains information about all systems on the network. Before a system can function with the mail program, you must modify the *sendmail.cf* file to identify the system. The file entry for a system includes the system's name and the transmission protocols it uses. You must enter the name of each system in lowercase letters. When *sendmail* is invoked to send a message, it uses the rules in *sendmail.cf* to resolve the *To* address of a particular system, user, and mailer (the program that transmits the message).

The *sendmail.cf* file is divided into sections by the type of transmission protocol. The IRIS Series 3000 workstation uses SMTP. List the other systems you want to send mail to using TCP/IP under the SMTP category. To add a system running SMTP, search the file for lines beginning with *CS* and add the system's name to it. For example, if you want to send mail to a system named *giants* that runs SMTP, edit *sendmail.cf* and search for *CS*.

The search takes you to these lines:

```
# Direct connect smtp hosts
CScardinals
CStigers
```

Add *giants*, with the prefix *CS* to the list:

```
# Direct connect smtp hosts
CScardinals
CStigers
CSgiants
```

If you do not send mail to certain systems, for example, *pirates* or *padres*, delete them from this list.

See the manual pages *sendmail*(1M), *mail*(1), and *Mail*(1) and the documentation within the file */usr/lib/sendmail.cf*.

## 2.7.2 Network Statistics

The *netstat* command provides statistical displays of network performance. *netstat* has the following options:

–A shows the address of any associated protocol control blocks. Use this for debugging.

–a shows the state of all sockets (normally, sockets used by server processes are not shown).

–i shows the state of interfaces that have been automatically configured. Interfaces that are statically configured into a system, but are not located at boot time, are not shown.

–m shows statistics recorded by the memory management routines (the network manages a private share of memory).

–n shows network addresses as numbers (normally *netstat* interprets addresses and attempts to display them symbolically).

–s shows statistics for each protocol.

–r shows the routing tables.

An example of the output of the *netstat* –r command is shown in Section 2.6.2.

### 2.7.3 Activating *rwho*

*rwho* is a network utility that tells you which machines are logged on the network. *rwho* can be used effectively in small networks with less than 20 systems. In large networks, *rwho* becomes a significant performance drain.

Find these lines in */etc/rc.tcp*:

```
#  Small  networks  can  use  rwho.
#  However, large sites, with > 20 rwho
#  machines can saturate the network.
#  In large networks, comment out the following 6 lines.
   if test -x /usr/etc/rwhod; then
        if test ! -d /usr/spool/rwho; then
            mkdir /usr/spool/rwho
        fi
   /usr/etc/rwhod;       echo " rwhod\c"
   fi
                 echo "."
```

If you do not want to use *rwho*, add comment marks to these lines of code so that it looks like the example below:

```
#  if test -x /usr/etc/rwhod; then
#       if test ! -d /usr/spool/rwho; then
#          mkdir /usr/spool/rwho
#       fi
#    /usr/etc/rwhod;      echo " rwhod\c"
#  fi
#                 echo "."
```

## 2.8 Setting your Internet Address

The IRIS has two network addresses, an Ethernet address and an internet address. The Ethernet address is a unique address that is burned into a PROM on the Ethernet board inside the IRIS. It never changes unless you change the Ethernet board. The internet address is a four-byte number and is determined by your network. Instructions for determining your network address are in Section 2.1.

It is good practice to set your internet address into your machine's PROM memory. Setting you internet address stores it in a special area of Random Access Memory (RAM) that has battery power backup, so that the contents are not lost when the machine is turned off or rebooted.

If you have PROMS at version 3.0.9 or later, follow this procedure.

1. Display the client's internet and Ethernet addresses at the PROM monitor. To get to the PROM monitor, type:

```
su
sync
sync
reboot
```

2. Determine whether you have already set your internet address into PROM memory. At the PROM monitor, type:

```
set
```

   This brings up a screen with information about your system, including network addressing. You must set your internet address if you see this message:

```
internet address of this machine: (not set)
```

To set your internet address, first, determine the internet address for your machine, (see Section 2.1). You must set the internet address of your machine locally using the PROM monitor *set* command. Type:

**set inetaddr** <*internet address*>

The third field in the command is the assigned internet address expressed in four decimal numbers, representing the four bytes of the 32-bit internet address separated by periods. Enter the number for the internet address assigned to your machine by the network or system administrator, (for example, 89.0.0.1).

3. If your internet address has been set into PROM memory, verify that it is correct with your network or system administrator.

Once you have used the `set inetaddr` command, you will not have to reenter it on subsequent reboots. If the internet address of your machine is ever changed, you must reset it using the `set inetaddr` command again.

# 3. Network Application Utilities

This chapter describes the six network application utilities included in the TCP/IP communications software:

| | |
|---|---|
| *rcp* | copies a file from one computer running UNIX to another computer running UNIX. |
| *rsh* | executes a command on a remote host running UNIX. |
| *rlogin* | initiates a login on a remote host running UNIX. |
| *rwho* | displays a list of the current users on remote UNIX hosts. |
| *ruptime* | displays the status of remote UNIX systems. |
| *Mail* | executes network mail. |

Chapter 4 describes the network applications utilities *telnet* and *ftp*. *rlogin* and *rcp* are more robust than *telnet* and *ftp*; if possible, use *rlogin* and *rcp*.

## 3.1 Using *rcp*, the Remote Copy Program

The address notation used by *rcp* has changed. Use the address notation below with GL2-W3.6 and subsequent software releases.

```
user@host
```

When using the new notation, make sure that your *stty* shell does not use the at sign (@) as an erase character.

*rcp* copies a file from one system to another. You specify the source machine, user, and the pathname for the file, followed by the destination machine, user, and the destination pathname for the file.

```
rcp [[user@]host:]pathname [[user@]destination:]pathname
```

The square brackets indicate that the information contained within them is optional. If you do not specify a name for either the source or destination machines, the system assumes the local machine.

Below are some examples for using *rcp*. In these examples, you must either have accounts on both hosts with the same user name or else specify the user's and host's name as shown below.

```
user@host
```

The user names on each machine must be equivalent to each other, through either */etc/hosts.equiv* or the user's *.rhosts* file.

### 3.1.1 Copying Files from Local to Remote Machines

This example copies the file *sqiral.c* in the current directory on the local machine to the file *sqiral.c* in the directory */oh4/doc/install* on a destination machine named *opus*. The system assumes that the local machine is the source of the file, because no machine is specified for the file *sqiral.c*.

```
rcp sqiral.c opus:/oh4/doc/install/sqiral.c
```

A *.rhost* file can help if you have difficulty transferring files to a specific directory in your home directory. See Section 2.4.3 for information on *.rhosts*. Follow this format to transfer files to a specific directory.

```
rcp filename user@host:~user/pathname
```

Another way to solve the file transfer problem is to copy the files to */tmp*.

```
rcp sqiral.c opus:/tmp
```

### 3.1.2 Copying Files from Remote to Local Machines

This example copies the file */usr/include/stdio.h* from the remote machine *sting* to the file *test* on the local machine. The system assumes that the local machine is the destination for the file, because no machine is specified for the file *stdio.h*.

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

### 3.1.3 Copying Files between Remote Machines

This example copies the file *temp_vi* from a remote machine named *puppy* to a file with the same name on a remote machine named *sting*. The system assumes that the user's directory exists on both machines since no home directory is specified.

```
rcp  puppy:temp_vi  sting:temp_vi
```

### 3.1.4 Copying Directory Trees

You can also use *rcp* to copy the entire directory tree from a remote machine to the local machine. The –r option causes recursive copying that copies the directory tree. In this example, all files and directories from */usr/include* on the remote machine *sting* are copied to the directory *localinclude* on the local host.

```
rcp  -r  sting:/usr/include  localinclude
```

Using *rcp* with the –v flag copies files in verbose mode, i.e., a list of the files being transferred is displayed on the monitor. This verifies that *rcp* is copying files.

## 3.2 Using *rsh*, the Remote Shell Program

*rsh* connects your system to a remote host and executes the commands you specify. Like *rcp*, this network utility assumes that you have accounts under the same user name on both the remote and local hosts. If you do not have an account on the remote machine, you can log in under another account name that you must prefix with `-l`. This is the *rsh* syntax:

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

For example, to find the load average on another machine, type:

```
rsh hostname -l pat uptime
```

If you do not specify a command for *rsh* to execute, *rsh* initiates an *rlogin* (remote login) on the remote machine. (See Section 3.3, below, on using *rlogin*.)

Interactive commands such as *vi*(1) do not run correctly when executed using *rsh*.

## 3.3 Using *rlogin*, the Remote Login Program

*rlogin* 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 *opus*, type:

```
rlogin opus
```

The specified remote system must be listed in the file */etc/hosts*. (See Section 2.4.1.)

If your system is listed in the remote host's */etc/hosts.equiv* file, you can *rlogin* to the remote host without a user name or password. Alternatively, you can provide an *.rhosts* file in your home directory on a remote machine. An *rhosts* file specifies an account on a host that is considered to be

equivalent to your local account. (See Section 2.4 for information on */etc/hosts.equiv* and *.rhosts*.) The *.rhost* file allows you to log in to a remote system as another user. In this example, the user logs in to the system *opus* as a user named *rick*.

```
rlogin opus -l rick
```

This example assumes that the user, *rick*, has an account on both the local and remote systems.

## 3.4 Using *rwho*

The *rwho*(1C) command generates a list of users 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 currently logged in. The list of hosts used to generate the names is obtained from the file */usr/spool/rwho*. See Section 2.7.3 for information on enabling *rwho*.

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

```
rwho
```

A list of users similar to the example below appears on the screen:

```
percy    opus:tty15     Mar 14 08:17
arnold   opus:ttyp4     Mar 14 08:35:25
root     knot:console   Mar 12 15:42
mary     knot: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 followed by the terminal name; the third column lists the login date and time as follows:

```
month day hour:minute:second
```

If a user has not typed to the system for a minute or more, *rwho* reports this idle time. If the user has not typed to the system for one hour or more, *rwho* does not display that user at all. To display a list of users that includes those who have not used their system for more than one hour, use the –a argument to *rwho*.

To obtain a list of users on a particular system, specify the host name as an argument to the *rwho* command. For example, to display a list of users on the host named *knot*, type:

```
rwho knot
```

## 3.5 Using *ruptime*

The *ruptime*(1C) utility displays the status of all UNIX hosts on the local network. Along with the name of the host, *ruptime* displays the current time, how long the system has been up, and the average number of jobs in the run queue for each system.

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

```
ruptime
```

The status list displayed is similar to this example:

```
opus      up 12+18:54      4 users, load 1.10, 0.83, 0.75
knot      down 18+13:25
percival  up 5+02:45       10 users, load 4.51, 3.87, 3.51
```

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. This 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 (for UNIX 4.3BSD systems only).

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

NOTE: *rwho* and *ruptime* do not work well in large networks or over gateways.

## 3.6 Using Mail

TCP/IP software extends the UNIX 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 command *Mail*, and the user's login name followed by the at sign (@) and the host machine. Use this notation:

```
Mail user@host
```

Then, type your message. Press CTRL-d to end the message. For example, to send mail to user *masa* who gets mail on the machine *edo*, address your message:

```
Mail  masa@edo
```

If you are sending mail to a user at a host on another network, you can use the *user@host* notation and put both the gateway machine's name and the destination machine's name in the address. The gateway machine is separated from the host machine by a period (.). For example, to send mail to user *masa* at host *edo*, which is linked to your network by the gateway machine *fuji*, type:

```
Mail  masa@edo.fuji
```

This address shows the machine *edo* is in the domain *fuji*. A domain is a hierarchical naming scheme that runs right to left. Depending on your network configuration, you may find it easier to use one of these address notations:

```
Mail  masa%edo@fuji
```

or

```
Mail  fuji!edo!masa
```

Talk with your system or network administrators to determine which address notation is most effective with your network. For information on configuring the files on your machine to enable network mail, see Section 2.7.1.

# 4. Using ARPANET Utilities

The utilities *telnet* and *ftp* are documented in this chapter. *telnet* and *ftp* are ARPANET-derived. This means they are commonly used in the ARPANET community. *rlogin* and *rcp* perform the same functions as *telnet* and *ftp*, and are faster and more reliable.

## 4.1 Using *telnet*, a Remote Login Program

Like *rlogin*, *telnet* initiates a login on a remote host across the network. Unlike *rlogin*, *telnet* uses the TELNET protocol for remote logins.

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

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

   ```
   telnet puppy
   ```

   The screen shows:

   ```
   Trying...,
   Connected to puppy.
   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 these options:

| | |
|---|---|
| ? [command] | get help on commands. |
| **status** | show status of *telnet*. |
| **escape** [esc-char] | change character for exiting *telnet*. |
| **close** | close a TELNET session. |
| **open** *host*_[port] | open connection on host. |
| **sendcrlf** | send a carriage return/line feed. |

See *telnet*(1C) for details.

## 4.2 Using *ftp*, the File Transfer Program

*ftp* is a program for transferring files using the ARPANET File Transfer Protocol. Below is a brief description of some of the *ftp* commands. For more information, see *ftp*(1C). See Section 4.2.2 for file transfer commands.

### 4.2.1 Entering and Exiting *ftp*

This procedure describes how to enter and exit *ftp*.

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

```
ftp hostname
```

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

```
ftp abbott
```

After you execute the above command, the screen shows:

```
Connected to abbott.
220 FTP server
    (Version 4.81 Mon Sep 26 08:36:28 PDT 1983)
ready.
```

The host then requests your name and password. This example assumes the user's name is *peter*.

```
Remote User Name: peter
331 Password required for peter.
```

Enter your password and the host replies:

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

After you have logged in, the *ftp* prompts appears:

```
ftp>
```

2. Now you are ready to send and receive files. See Section 4.2.2 below for examples of file transfer commands. To see a list of the commands for *ftp*, type:

```
help
```

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

```
quit
```

If you frequently copy files from a remote host using *ftp*, you can create a *.netrc* file in your home directory to allow you to log in without a password.

To create *.netrc*, edit it so that it contains the name of the remote machine, your user id, and your password. After you create it, use *chown* to make *.netrc* readable only by you. Below is a sample *.netrc* file.

```
machine  machine_name
user user_name
password  password_on_machine
```

### 4.2.2 *ftp* Commands

Each section below describes some of the commands that you can use with *ftp*. Other commands are available; see *ftp*(1C) for more information. Both the command syntax and an example for that command are given.

#### Transferring Files from Local to Remote Machines

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

**send** *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:

```
send 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:

```
200 PORT command okay.
150 Opening Data connection for myfile (47.0.0.131,1601)
226 Transfer complete.
21  bytes send in 0.031 seconds (0.62 Kbytes/s)
```

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

**append** *localfile remotefile*

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/john/filexfer*, type:

```
cd /usr/john/filexfer
mput thisfile thatfile
```

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

## 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.

Another command for obtaining a file from a remote machine has this syntax:

```
recv remotefile [localfile]
```

For example, to get the file *sqiral.c* from the remote machine and give it the same name on the local machine, type:

```
recv sqiral.c
```

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
```

The screen indicates that the files have been received.

## Executing Local Commands

To invoke commands on the local machine, precede the command with an exclamation point (!). For example, to see if the file that you sent to the local machine has been received, 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
```

## 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*]

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.

## Transferring Binary or ASCII Files

You can transfer either binary or ASCII files with *ftp*. By default, *ftp* transfers ASCII files. 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 work identically for either ASCII or binary file transfer.

# 5. Network Connections Within a Program

This chapter describes three programs, *accept*, *select*, and *connect*. These programs are examples of how to make a TCP/IP connection between machines from within a program. A listing of these programs is given in this chapter.

To compile and run *accept*, *select*, and *connect*, follow the instructions below:

1. Create the *Makefile* by typing in the text below. See *make*(1).

```
all:    accept connect select

accept: accept.c
        cc -O accept.c -o accept -lbsd

connect:connect.c
        cc -O connect.c -o connect -lbsd -ldbm

select: select.c
        cc -O select.c -o select -lbsd

clean:  rm -f accept connect select
```

2. Compile the files *accept.c*, *select.c*, and *connect.c* with the *Makefile*. This builds the executable files *accept*, *select*, and *connect*. To compile *accept.c*, *select.c*, and *connect.c*, type:

```
make
```

3. Run *accept* on one machine; then run *connect* on another machine. Everything you type on the connect side appears on the accept side.

For example, at site A, type:

```
accept
```

At site B, type:

```
connect  hostA
this  is  a  message
```

Back at site A, the following appears on your screen:

```
this  is  a  message
```

4. To end the session, type:

```
EOF
CTRL-d
```

5. You can also run *select* on one machine and run *connect* on another machine. For example, at site A, type:

```
select
```

At site B, use the command *connect* with the name of the machine to which you want to connect. For example, at site B, type:

```
connect  hostA
```

At site A, type a message:

```
This  is  a  message.
```

The message appears:

```
This  is  a  message.
sending  <This  is  a message.\n>
```

Back at site A, the following appears on your screen:

```
This is a message.
```

To end the connection at site B, press CTRL-d.

On site A, you see a message:

```
waiting for a connection
```

This message repeats until a *connect* message is received from site B again, another machine or until you press CTRL-c.

**NOTE:** The programming interface for the TCP/IP software is based on UNIX 4.3 BSD.

# 5.1 The *accept* Program

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

main()
{
   int sock, length;
   struct sockaddr_in sin;
   int msgsock;
   char line[80];
   int i, cnt;

/* create a socket */

   if ((sock = socket (AF_INET,SOCK_STREAM,0)) < 0) {
      perror("opening stream socket");
      exit(0);
      }

/* initialize socket data structure */

   sin.sin_family = AF_INET;
   sin.sin_addr.s_addr = INADDR_ANY;
   sin.sin_port = htons(IPPORT_RESERVED + 1);

/* bind socket data structure to this socket */

   if (bind (sock,&sin,sizeof(sin))) {
      perror("binding stream socket");
      }

/* getsockname fills in the socket structure with information,
   such as the port number assigned to this socket */
```

```
    length = sizeof(sin);
    if (getsockname (sock,&sin,&length)) {
       perror("getting socket name");
       exit(0);
       }
    printf("Socket has port# %d\n",htons(sin.sin_port));

/* prepare socket queue for connection requests and accept
    connections */

    listen(sock,5);
    if ((msgsock = accept(sock,0,0)) <= 0) {
       perror("accept on socket");
       exit(0);
       }

/* read from the message socket and write to standard out */

    while ( (cnt = read(msgsock, line, 80)) > 0)
       write(1,line,cnt);

    close(msgsock);
    printf("Done\n");
}
```

## 5.2 The *connect* Program

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


main(argc,argv)
int argc;
char **argv;
{
   int cnt,sock;
   struct sockaddr_in sin;
   struct hostent *hp, *gethostbyname();
   char line[80];

   if (argc != 2) {
      printf("usage:%s host0,argv[0]);
      exit(0);
   }

/* open socket */

   if ((sock = socket (AF_INET,SOCK_STREAM,0)) < 0) {
      perror("opening stream socket");
      exit(0);
      }

/* initialize socket data structure */

   sin.sin_family = AF_INET;
   hp = gethostbyname(argv[1]);     /* to get host address */
   bcopy (hp->h_addr, &(sin.sin_addr.s_addr), hp->h_length);
   sin.sin_port = htons(IPPORT_RESERVED+1);
```

```
/* connect to remote host */

   if (connect(sock,&sin,sizeof(sin)) < 0) {
      close(sock);
      perror("connection streams socket");
      exit(0);
      }

   while ((cnt = read(0,line,80)) > 0) {
      printf("sending <");
      fflush(stdout);
      write(1,line,cnt-1);
      printf("\n>\n");
      fflush(stdout);

/* send input to remote host */

      if (write(sock,line,cnt) < 0) {
         perror("writing on stream socket");
         exit(0);
      }
   }
   printf ("Done\n");
   close(sock);
}
```

## 5.3 The *select* Program

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <bsd/netdb.h>
#include <stdio.h>
#include <time.h>
#define TRUE 1

main()
{
   int sock, length;
   struct sockaddr_in sin;
   int msgsock;
   char line[80];
   int ready, i, cnt;
   struct tm to;

/* create a socket */

   sock = socket (AF_INET,SOCK_STREAM,0);
   if (sock < 0) {
      perror("opening stream socket");
      exit(0);
      }

/* initialize socket data structure */

   sin.sin_family = AF_INET;
   sin.sin_addr.s_addr = INADDR_ANY;
   sin.sin_port = htons(IPPORT_RESERVED + 1);
```

```
/* bind socket data structure to this socket */

   if (bind (sock,&sin,sizeof(sin))) {
      perror("binding stream socket");
      }

/* getsockname fills in the socket
   structure with information, such as the
   port number assigned to this socket */

   length = sizeof(sin);
   if (getsockname (sock,&sin,&length)) {
      perror("getting socket name");
      exit(0);
      }

/* prepare socket queue for connection requests
   and accept connections */

   listen(sock,5);
   do {
      ready = 1<<sock;
      /* accept connections requests on this socket */
      to.tm_sec = 5;
      select(20,&ready,0,0,&to);
      /* are there any requests ? */
      if (ready) {
         msgsock = accept(sock,0,0);
         while ( (cnt = read(msgsock, line, 80)) > 0)
            write(1,line,cnt);
      close(msgsock);
      }
      else printf("Waiting for connection\n");
   } while (TRUE);

   printf("Done\n");
}
```

**Silicon Graphics, Inc.**

Date _____

Your name _____

Title _____

Department _____

Company _____

Address _____

_____

_____

Phone _____

**COMMENTS**

Manual title and version _____

_____

Please list any errors, inaccuracies, or omissions you have found in this manual

_____

_____

_____

_____

Please list any suggestions you may have for improving this manual

_____

_____

_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 45 MOUNTAIN VIEW, CA

**POSTAGE WILL BE PAID BY ADDRESSEE**

**Silicon Graphics, Inc.**
Attention: Technical Publications
2011 Stierlin Road
Mountain View, CA 94043-1321