SANYO

ICON

ICON/UXV-NET Networking Tools Guide

ICON
INTERNATIONAL

764 East Timpanogos Parkway
Orem, Utah 84057-6212
(801) 225-6888

TOOLS GUIDE

ICON/UXV-NET Networking Software

Order No. 172-054-001 A1

Trademarks

Record of Changes

ICON/UXV-NET
Administrator's Guide & Programmer's Reference Manual
Order No. 172-054-001

Date	Update†	Change	Entered By *
Sep 1988	A0	Initial publication of Revision A	
Mar 1989	A1	Addition of new manual pages and appendices F and G	

† An update number has two parts: a capital letter and an Arabic numeral. (See update number A0 above.) The capital letter refers to the revision of the manual and the Arabic numeral refers to the sequence of changes made to that particular revision.

The first publication of all manuals is always designated as Revision A and is presented as A0. After the number of changes made to a particular manual warrants a new edition, the revision letter is changed to the next capital letter. For example, the revision after Revision A will be Revision B, and the publication will be represented as B0.

The second part of the update number, the Arabic numeral, gives the consecutive order of changes made to each revision. Update number A1 represents the first change made to Revision A, update A2 is the second change, and so forth. When a new revision is issued, the numbering starts over (B0, B1, B2).

* The person who entered the updated pages into this manual.

SANYO
ICON

Dear Customer,

The enclosed Update Package contains updated replacement pages for the ICON/UXV-NET Networking Tools Guide, formerly known as the ICON/UXV-NET Administrator Guide.

Instructions are provided to help you insert the replacement pages. There is also a new "Record of Changes" page that provides you with a history of the changes that have been made to this manual since it was first published.

If you have any questions concerning this update to your ICON/UXV-NET Networking Tools Guide, or any previous updates (as you review the change history on the "Record of Changes" page), please contact the Sanyo/Icon Customer Service Department toll free at 1-800-777-ICON.

Sincerely,

Stephen Raff

Stephen Raff
Technical Publications

ICON INTERNATIONAL, Inc.
764 East Timpanogos Parkway
Orem, UT 84057-6212
1-800-777-ICON
Fax: 801 226-0651
Telex: 323938 ICONSYS

Change Instructions

ICON/UXV-NET
Networking Tools Guide
Order No. 172-054-001
Update A1

Update this manual as follows:

1. Remove the old pages and insert the new pages as indicated below.
2. For future reference, file these instructions in front of the manual after "Record of Changes."

NOTE: All holders of this manual should incorporate these changes into their copies.

Remove Old Pages	**Insert New Pages**
Front Cover	Front Cover
Front Matter, comprising the Title Page and pages ii – vi, which includes the Copyright page, Record of Changes page, and Table of Contents	Front Matter, comprising the Title Page and pages ii – vi, which includes the Copyright page, Record of Changes page, and Table of Contents
Section 1, "Introduction", pages 1-1 through 1-4	Section 1, "Introduction", pages 1-1 through 1-4
Section 2, "Installation and Power Up", pages 2-3 through 2-6	Section 2, "Installation and Power Up", pages 2-3 through 2-6
All of Section 3, "Configuration and Maintenance"	All of Section 3, "Configuration and Maintenance"
Section 4, "Network Setup", pages 4-3 through 4-6	Section 4, "Network Setup", pages 4-3 through 4-6
The "Appendices" preface, pages A-i and A-ii	The "Appendices" preface, pages A-i and A-ii
All of Appendix A, "Manual Pages"	All of Appendix A, "Manual Pages"
	Appendix F, "An Introductory Interprocess Communication Tutorial"
	Appendix G, "An Advanced Interprocess Communication Tutorial"
Back Cover	Back Cover

DONE
EL 3.21.89

Table of Contents

Introduction 1

The ICON/UXV-NET product enables your ICON system running ICON/UXV to use a subset of the networking utilities originally developed for use at the Advance Research Projects Agency (ARPA) and the University of California at Berkeley (UCB). The utilities that originate from UCB are based on Berkeley's Software Distribution of UNIX®, version 4.3 (4.3BSD).

These networking utilities enable you to transfer files, log into remote hosts, execute commands remotely, and send mail to and receive mail from remote hosts on the network.

The *ICON/UXV-NET Administrator's Guide* provides specific information about the operation and maintenance of the ICON/UXV-NET networking utilities for the ICON 2000, 3000, 4000, and 5000 systems. This manual also provides detailed information about the internetwork mail routing facility provided with this product, as well as supplementary documents and "manual pages" that will prove useful to the network administrator.

Manual Overview

Who Should Read This Manual

This manual is intended to cover all aspects of the ICON/UXV-NET Networking Tools. The following areas of interest are covered:

- network software installation and configuration
- routine network administration
- operation of network commands
- use of the networking software development library

A working knowledge of ICON/UXV commands and directory structures as well as the ability to become a super-user and manipulate files with an editor, such as *vi*, is required. You should have access to the ICON/UXV Reference Manuals and Guides and be comfortable in your knowledge of the ICON/UXV operating system. You should also be familiar with the Operator's Manual for your ICON computer system.

In the case of Ethernet® networking, familiarity with board installation and removal procedures is necessary. This manual assumes that ICON/UXV, version 3.30 or later, has been installed on the ICON systems which are networked.

What Is In This Manual

The list that follows briefly describes the contents of each section and appendix in this manual.

Section 1: Introduction

The remaining part of this section provides a list of reference manuals that you may need and a quick-reference list of the daemons, libraries, security and configuration files that will help you manage the ICON/UXV-NET utilities and services. Obtaining an Internet domain name and obtaining information about Request for Comment documents (RFC) and Military Standards (MIL-STD) is also explained.

Section 2: Installation

This section describes the installation of the ICON/UXV-NET utilities on an ICON system. Information is also presented about maintaining a system networking map and the parts that make up the Ethernet and Serial-Link network connections.

Section 3: Configuration and Maintenance

This section describes how to configure and maintain the ICON/UXV-NET network. Also presented is a discussion on the files that must be altered to properly configure your system for networking.

Section 4: Network Setup

This section provides general information on setting up networks using the ICON/UXV-NET networking environment and an ICON computer system.

Appendix A – Manual Pages

This appendix provides the documentation for each of the utilities supported in the ICON/UXV-NET product. These pages are the same as the manual pages that are electronically on-line in your ICON system.

Appendix B – SENDMAIL – An Internetwork Mail Router

This document describes *sendmail*, the internetwork mail routing facility provided with the ICON/UXV-NET product. Included are guidelines for deciding whether to install *sendmail*, details about *sendmail* and its configuration file, installation instructions, and guidelines for modifying the supplied *sendmail* configuration file.

Appendix C – SENDMAIL Installation and Operating Guide

This document describes how to install and operate a basic version of SENDMAIL, the Internetwork Mail Routing program. It is a logical extension of the document found in Appendix B.

Appendix D – Introduction to the Internet Protocols

This document provides an introduction to the facilities and capabilities of the Internet Protocols. Information is provided that describes other documents, referred to as "RFC" and "IEN" documents, and how to obtain a copy of those documents.

Appendix E – Networking Implementation Notes

This document describes the internal structure of the networking facilities.

Appendix F – An Introductory 4.3BSD Interprocess Communication Tutorial. This document describes the use of pipes, socketpairs, sockets, and the use of datagram and stream communication. The intent is to present a few simple example programs, not to describe the networking system in full.

Appendix G – An Advanced 4.3BSD Interprocess Communication Tutorial. This document provides an introduction to the interprocess communication facilities included in the ICON/UXV operating system, discusses the overall model for interprocess communication, and introduces the interprocess communication primitives which have been added to the system. A working knowledge of the C programming language is expected as all examples are written in C.

Managing the Network

The daemons (server processes that run continuously, in the the background, to provide services to users), servers, configuration files, user and administration programs that will help you manage the ICON/UXV-NET utilities are briefly described in the following quick-reference list.

Daemons

/etc/inetd	master server process; initiates servers below
/etc/routed	network route information server
/etc/rwhod	network user information server
/usr/lib/sendmail	mail server, network mail router, local mail delivery
/etc/syslogd	system error log facility

Servers

/etc/ftpd	File Transfer Protocol (*ftp*) server
/etc/remshd	remote shell server
/etc/rexecd	remote program execution server
/etc/rlogind	remote login server
/etc/telnetd	DARPA Telnet Protocol server
/etc/ntalkd	new talk protocol server
/etc/talkd	old talk protocol server
/etc/tftpd	Trivial File Transfer Protocol (tftp) server

Configuration Files

/usr/lib/aliases	mail alias data base
/etc/ftpusers	if present, contains list of users allowed to use *ftp*
/etc/gateways	routing information to gateway hosts
/etc/hosts	internet host address table (remote hosts)

Configuration Files (Continued)

/etc/hosts.equiv	list of "trusted" hosts
/etc/inetd.conf	inetd configuration file
/etc/networks	networks known to this host
/etc/protocols	protocols known to this host
/etc/services	services available through inetd
/etc/syslog.conf	syslogd configuration file

User Programs

ftp	File Transfer Protocol program
mail	network mail program
rcp	network copy program
rdist	remote file distribution program
remsh	run a command on a remote host
rlogin	login to a remote host
ruptime	provides information on length of time remote hosts have been up, how many users, and host load average
rwho	who is logged in on remote hosts
talk	converse with a local or remote user
telnet	user interface to the TELNET protocol
tftp	Trivial File Transfer Protocol program

Administration Programs

gettable	get NIC format host tables from a host
htable	conver NIC standard format host tables
ifconfig	configure network interface parameters
netstat	show network status
newaliases	rebuild the data base for the mail aliases file
ping	test availability of other network hosts
route	manually manipulate the routing tables
slattach	attach serial lines as network interfaces
trpt	transliterate protocol trace

Reference Manual Guide

For more information on the following subjects, refer to the publications listed in the right column.

For Information On:	**Read:**
ICON/UXV Administration	*ICON/UXV Administrator Guide* *ICON/UXV Release Documentation Package*
Installing Ethernet hardware on ICON systems	*ENET Controller Board* in the *ICON System Reference Manual*
C Programming Language	*The C Programming Language,* Brian W. Kernighan, Dennis M. Ritchie; Prentice-Hall, Inc. *C Programming Guide,* Jack Purdum, Que Corporation, Indianapolis, Indiana
ICON/UXV operating system	*ICON/UXV User Guide* *ICON/UXV User Reference Manual* *ICON/UXV Editing Guide* *ICON/UXV Administrator Guide* *ICON/UXV Administrator Reference Manual* *ICON/UXV Programmer Guide* *ICON/UXV Programmer Reference Manual* *Exploring the UNIX System,* Kochan & Wood, Hayden Book Company

Details on the various protocols used in ICON/UXV-NET are discussed in technical publications know as "Request For Comments". The following is a partial list of those publications.

For Information On:	**Read:**
Address Resolution Protocol	RFC 826
Domain Requirements	RFC 920
File Transfer Protocol	MIL-STD 1780 RFC 959
Internet Control Message Protocol	RFC 792
Internet Protocol	MIL-STD 1777 RFC 791
Simple Mail Transfer Protocol	MIL-STD 1781 RFC 821

Standard for the Format of ARPA Internet Text Messages	RFC 822
Telnet	MIL-STD 1782

To obtain information about available RFCs, contact:

Network Information Center (NIC)
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
1-800-235-3155

To obtain information about available MIL-STD specifications, contact:

Department of the Navy
Naval Publications and Forms Center
5801 Tabor Avenue
Philadelphia, PA 19120-5099

Installation 2

Introduction

Networking in ICON computer systems is implemented with standard hardware and software configurations for maximum compatibility in a multi-vendor environment. Two interfaces are currently supported: Ethernet and Serial-Link. ICON machines can support both interfaces simultaneously if desired. The TCP/IP protocol standard is used on both Ethernet and Serial-Link connections.

This section includes information on how to:

- set up and maintain a network map;
- make sure the proper hardware is installed on your computer system;
- install the ICON/UXV-NET software; and
- add your computer system to the network.

Overview of Installation

The list below is an overview of the installation procedure described in this section. To install and power up the ICON/UXV network on your ICON computer system:

- set up a network map;
- make sure the network hardware is properly installed;
- install the ICON/UXV network software;
- choose an internet address and host name;
- assign host names and internet addresses.

System Preparation

Preparing your ICON system for networking requires setting up and maintaining a network map and installing the necessary networking hardware within your system's configuration. The rest of this section describes what you must do to set up your ICON computer system to operate on the network.

Maintaining a Network Map

As you install a new computer node on your network, it is important to take the time to update your network map. If you have not previously created a map, it is strongly recommended that you create one now. A network map provides you with information about the location and configuration of the computers on the network. As network administrator, it is your responsibility to keep the network map up-to-date when you add or delete computers or make cable changes.

Your network map should contain the location of:

- the coaxial cable, including terminators and repeaters;
- the network Transceiver or Fan-out Unit and Controller cables;
- the taps into the coaxial cable;
- the networking equipment; and
- each node on the network, including the:
    - complete node (host) name;
    - internet address;

The example shown on the next page is a sample network map using both Ethernet and Serial-Link hookups.

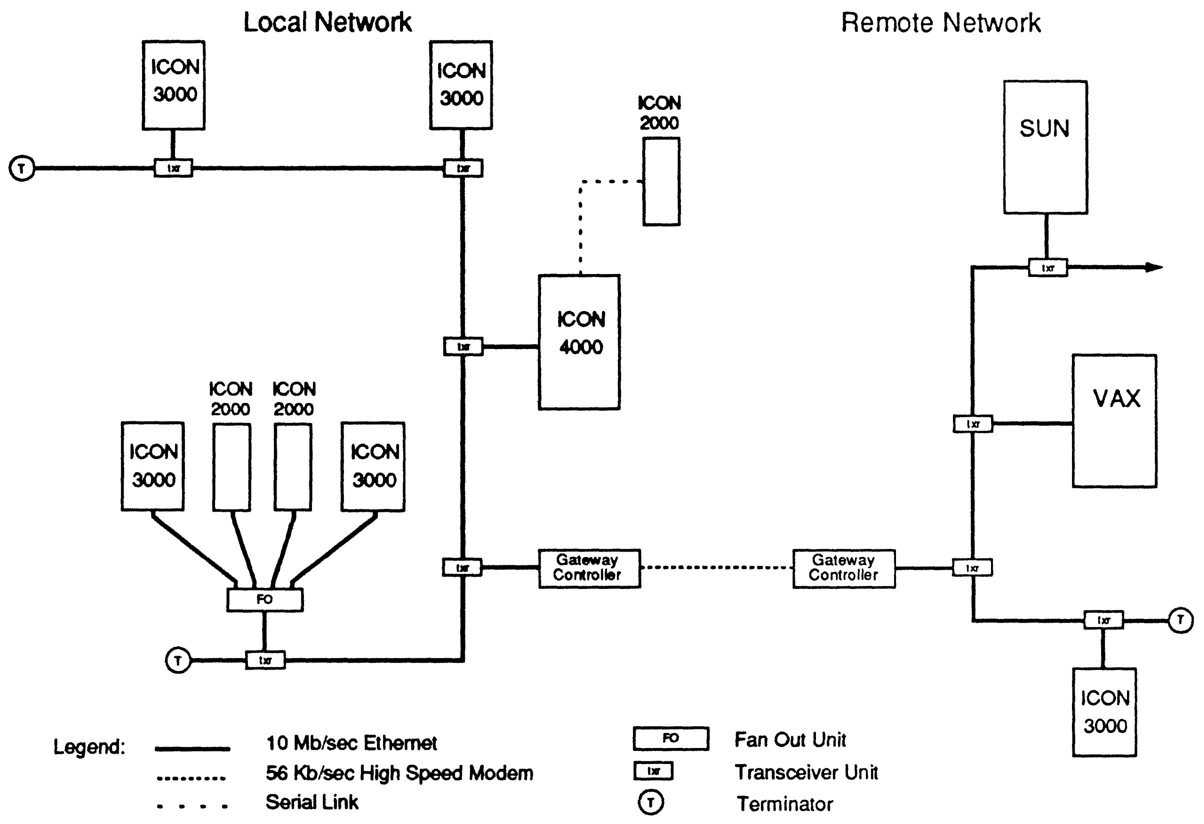

Figure 1. *Sample Network Map*

Preparing Your ICON System for Networking

Several components are required to prepare your ICON computer system for operation on the network. Those components include hardware for Ethernet and Serial-Link connections and networking software.

Setting up hardware for networking consists of configuring jumpers on boards and installing boards and cables in the proper locations within the computer system configuration.

Ethernet Connections

The Ethernet connection for networking is made by installing the ICON ENET option (PN 840-026-001), which includes:

- MBA1 (Multibus Adapter) Board
- ENET Controller Board
- ENET Controller Cable
- ENET Transceiver Cable - 33 feet long
- ENET Transceiver (for connecting to N-type coaxial cables), or
- ENET Transceiver Fan Out Unit (for connecting up to 8 hosts on a single hub)

For directions on how to install the MBA1 and ENET Controller boards in your ICON system and how to correctly configure the necessary jumper settings, refer to the following manuals:

• *Multibus Adapter Board MBA1*	PN 170-022-001
• *ENET Controller Board*	PN 170-024-001

Serial-Link Connections

Serial-Link connections for networking use conventional RS-232C cables between serial ports on the Peripheral Communications Processor (PCP16) board in an ICON system. Serial-Link networks can be configured in various topologies such as stars or rings, with the understanding that each physical link between two machines is considered a separate network and that a maximum of five Serial-Links can be connected to any one machine. Computer systems that are not directly connected may still communicate through any number of gateway machines. To avoid line-ringing problems, it is recommended that inter-machine serial connections not be made until the ports that are to be connected are made non-login ports by changing the entries for those ports in the */etc/inittab* file. Systems with PCP16 boards installed should use the PCP serial ports rather than any port on the CPU board to obtain the highest possible performance.

Serial-Link components include:

- PCP16 Processor Board
- RS-232C Software Handshake Cable

NOTE: Although the DCS (Distributive Communication System) option provides additional serial and parallel communication ports in an ICON computer system, it is not recommended to be used for Serial-Link connections because of a noticeable degradation of communication performance.

Software Installation Information

Your ICON system must have version 3.30 of the ICON/UXV operating system (or later) installed and running before the ICON/UXV-NET software can properly be installed. ICON/UXV-NET networking software, version 3.30 or greater, provides all of the necessary utilities to implement networking under the ICON/UXV operating system on your ICON computer system.

You will need an internet address number for each host on your network to properly configure your network. Unregistered, but reserved Class C internet address are available from:

Icon Customer Service
Internet Network Address Coordinator
764 East Timpanogos Parkway
Orem, UT 84057

Registered internet addresses are available from:

Network Information Center
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

Unless you plan on connecting your netwrok directly to other TCP/IP networks, the default internet addressess supplied with ICON/UXV-NET are adequate.

Installing ICON/UXV-NET Software

The software for ICON/UXV-NET is in *tar* format. Once the release media is loaded into the appropriate drive mechanism, log in to the operating system *root* account (in response to the "`login`" prompt), by entering:

```
login:  root
```

The *login* program will prompt you to enter the password to the *root* account. After entering the password and receiving the *login* information, extract the ICON/UXV-NET software from the release media into the *root* account by entering:

```
cd /                    (positioned in root account)
tar  xvp                (extract software files from tape)
```

The contents of the release media will be extracted into the root account and all appropriate permissions will be assigned. When the extraction, installation, and configuration process is complete, you may log out of the *root* account and log back into your user account.

Configuring the System for Networking

Once the ICON/UXV-NET software is on your system, it is necessary to configure your system for networking. Chapter 3, *Configuration and Maintenance*, explains in detail the necessary steps to take so that networking will function properly on your ICON system.

Configuration and Maintenance 3

Introduction

This section describes how to configure and maintain your ICON/UXV network, including:

- how to set up the configuration files;
- a description of network daemons and servers;
- how the security algorithms work for each network service;
- guidelines on network connections; and
- how to perform maintenance tasks.

Software Configuration

The discussion that follows describes the steps necessary to configure the ICON/UXV-NET software on an ICON computer system for networking. The Ethernet and Serial-Link interfaces are discussed together because much of the software configuration applies to both and in many instances both will be present on the same machine. The files discussed below should be altered only by a system administrator with *super-user* privileges on each of the machines to be connected. Files can be changed while others are using the machines. But ultimately, shutdown and reboot of each machine in the network configuration will be necessary as a final step to bring up the networks.

Topology

The figure below shows a sample topology of a network configuration for the purposes of this discussion. Three systems, with hostnames ***doc***, ***sleepy***, and ***dopey***, have connections to an Ethernet network. In addition, ***doc*** has Serial-Link connections to ***grumpy*** and ***sneezy***. ***Sneezy*** has, in turn, a Serial-Link to ***happy***.

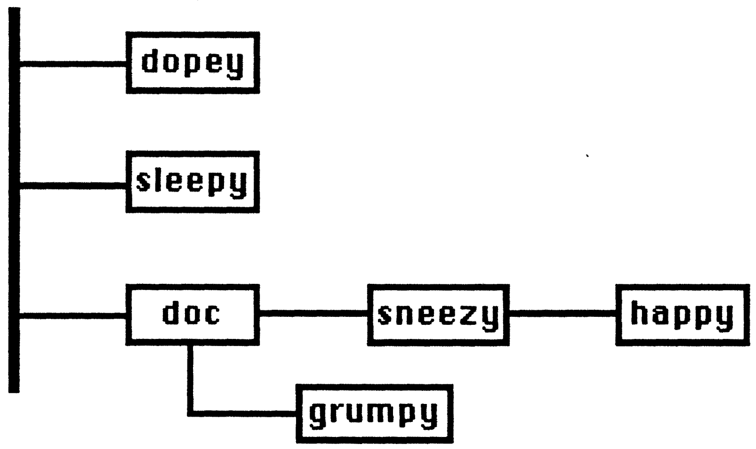

Figure 2. *Sample Topology*

The /etc/hosts File

Each of the six hosts in our sample topology has an */etc/hosts* file which specifies the internet address chosen for each host, its name, and an indication of which system is the ***loghost*** (which is simply the name of the host on which the file resides). Each of the six hosts should have identical information in the */etc/hosts* file except for the specification of the loghost. The following is an example */etc/hosts* file for **doc** :

```
#
# Example hosts file
#
192.41.100.1   dopey
192.41.100.2   sleepy
192.41.100.3   doc        loghost
192.41.99.1    grumpy
192.41.98.1    sneezy
192.41.97.1    happy
127.1          localhost
#
```

Except for the `localhost` entry, each host has a Class C internet address composed of four numeric values separated by periods. The leftmost three values together (e.g., 192.41.100) comprise the network number and the rightmost value is the host number.

In the Class C internet address schema, the network number may range from 192.1.1 to 223.254.254 and the host number may range from 1 to 254. If more than 254 hosts are required on a given network, class B internet addresses may be used. In this case, the leftmost two values are the network number and range from 128.1 through 191.254, while host numbers range from 1.1 to 254.254.

The network portion of an internet address corresponds to the network to which that host is primarily connected. Each host has only one internet address, even if it has connections to several other networks. In our sample case, network number `192.9.200` is the Ethernet connection between ***doc***, ***sleepy***, and ***dopey***. Each of the systems has the same network number (`192.41.100`) but different host numbers, ***dopey*** being "`1`", ***sleepy*** being "`2`", and ***doc*** being "`3`".

Each Serial-Link connection constitutes a separate network and therefore must have a separate network number. Network number `192.41.99` is assigned to the connection between ***grumpy*** and ***doc***. It has been chosen as ***grumpy 's*** primary connection, so the internet address for ***grumpy*** is "`192.41.99.1`". Network number `192.41.98.1` is assigned to the connection between ***sneezy*** and ***doc***, and appears in ***sneezy 's*** address. Network number `192.41.97.1` is assigned to the connection between ***happy*** and ***sneezy*** and appears in the address for ***happy***.

The network number also appears in the */etc/networks* file. Each hostname should appear only once in the */etc/hosts* file. Keep in mind that internet addresses assigned can be arbitrary within the above constraints unless the host is attached to the "official" internet. In that case, addresses must be obtained from the controlling authority. The entry for "`127.1 localhost`" is always present in the */etc/hosts* file.

The /etc/networks File

Each host must have an */etc/networks* file which specifies the network names, the internet addresses, and the networks that are directly accessible to that host. Each host system */etc/networks* file should have identical information except for the indications of directly accessible networks via the localnet specifier. A sample */etc/networks* file for ***sneezy*** would be as follows:

```
#
#  Example networks file
#
loopback    127
doc-ether   192.41.100
sl-grumpy   192.41.99
sl-sneezy   192.41.98    localnet
sl-happy    192.41.97
#
```

Note that the internet address for each network corresponds to the values in the */etc/hosts* file. In this case, there are two localnet specifiers because ***sneezy*** can directly access two of the Serial-Link networks, one to ***happy*** and one to ***doc***. Similarly, the /etc/networks file on ***doc*** will have three localnet entries and ***dopey*** will have only one. The network names are arbitrarily chosen strings without embedded spaces. The "`loopback 127`" entry is always present in the */etc/networks* file.

The /etc/gateways File

The */etc/gateways* file is only necessary for systems having Serial-Link connections; Ethernet connections do not require this file. It is required for Serial-Links because currently it is not possible to share routing information across Serial-Link interfaces as it is with Ethernet. Thus, your ICON system has no way of determining which Serial-Link interface to use to get to another host. The routes are established when the system boots and the *routed* daemon begins execution. The */etc/gateways* file is read by routed to build routing table entries for Serial-Link connections.

There are two kinds of entries in the */etc/gateways* file, those for hosts and those for networks. They are distinguished by the first word in the entry, either "`host`" or "`net`". Host entries are used to establish a route to a particular host and net entries establish routes to networks. A host entry is of the form:

host hostname **gateway** gatename **metric** hopcount **passive**

"`Hostname`" is the name of the destination host, "`gatename`" is the name of the host gateway, and "`hopcount`" is a value (0,1,2...) which indicates how many "hops" to the destination host. This number does not need to be accurate, as it is only used for routing path decisions in the case of multiple available paths to a destination. If the hopcount total exceeds 16, however, the destination will be considered "too far" and connection attempts will be abandoned.

Systems with Serial-Link connections must have a host entry in their */etc/gateways* file for each non-primary Serial-Link connection. A non-primary connection is defined as any Serial-Link connection to that host which has a network number in the */etc/networks* file which is different than the network number portion of the host's internet address as found in the */etc/hosts* file. Thus, a system with only one Serial-Link connection does not require a host entry in this file because the only connection it will have will be a primary connection. An example of this in our sample topology is the connection to ***happy*** (see Figure 2). The primary connection for ***sneezy*** is the link to ***doc***, so the */etc/gateways* file on ***sneezy*** must have a host entry only for ***happy***. Routing information for a primary connection is established with the **ifconfig** command in the */etc/rc.local* file discussed later.

The "`gatename`" is the name of the directly connected host which will act as a gateway to get to the destination. If the destination is a directly connected host, as is most common, the "`gatename`" is the name of the host on which the */etc/gateways* file resides. In other words, you must "gateway" through "yourself" to get to an adjacent host on a non-primary network.

Net entries are of the form:

```
net netname gateway gatename metric hopcount passive
```

"Netname" is the name of the destination network as defined in the */etc/networks* file and "gatename" and "hopcount" are as defined above. Net entries inthe */etc/gateways* file are typically only used to establish a default routing path. This can reduce the amount of information that must be expressed in */etc/gateways.* The default routing path will be used for any destination hosts or networks not explicitly defined in */etc/gateways.* A default path is established with a net entry using the value 0 for "netname" and an accessible host to use as the gateway in "gatename". Systems with only a single Serial-Link will usually have a single net entry of this form in their */etc/gateways* file. An example of this is *grumpy*. The following is a sample */etc/gateways* file for ***grumpy*** :

```
net 0 gateway doc metric 1 passive
```

A more complicated set of entries is required for ***doc*** , as shown below:

```
host grumpy gateway doc metric 0 passive
host sneezy gateway doc metric 0 passive
host happy gateway sneezy metric 1 passive
```

One more example is shown from ***sneezy*** :

```
host happy gateway sneezy metric 0 passive
net 0 gateway doc metric 1 passive
```

The /etc/hosts.equiv File

The */etc/hosts.equiv* file is simply a list of known and recognized hosts. These hosts will be allowed to **rlogin** and exchange other services, such as **rcp**, without password checking if an equivalent username is found on the local host. In our example network, all systems are allowed these privileges, so the */etc/hosts.equiv* files on all six machines are the same:

```
doc
sleepy
dopey
grumpy
sneezy
happy
```

If allowing equivalent usernames free access between machines is not acceptable, password checking for rlogin may be invoked by leaving the source hostname out of the destination's */etc/hosts.equiv* file. The root user is always restricted from performing rcp commands and is password checked with **rlogin** in every case.

After the */etc/hosts.equiv* file is in place, running the **MAKEHOSTS** script in */usr/hosts* will create a set of symbolic links to **remsh** for each host. By placing the file */usr/hosts* in your search path, remote execution can be invoked by simply typing the hostname on the command line, as in:

```
doc troff -ms < myfile.ms
```

This command invokes a "troff" on ***doc*** using "ms" macros with input from *myfile.ms* on the local host. An **rlogin** can be invoked with just the hostname, as in:

```
doc
```

The /etc/inittab File

The */etc/inittab* file must be configured so that all ports connected to Serial-Link networks are non-login ports. This is done by changing the third field (fields are separated by ":" (colon) characters) in the entry for the port to "off", as in:

```
14:2:off:/etc/getty ttyaa 9600 dt1200 # PCP 0 line 10
```

which prevents a login from being allowed on port `ttyaa`.

The /etc/rc.local File (For ICON/UXV Software Release 3.3X Only)

The */etc/rc.local* file must contain commands which initialize network interfaces when the system boots. Serial-Link interfaces each require two lines in this file and Ethernet interfaces require one. The following is a sample */etc/rc.local* file entry for *doc* :

```
# example /etc/rc.local for doc
# Don't forget to change "/etc/hosts" if you change your hostname!
hostname  doc
/etc/ifconfig lo0 localhost
/etc/route add `hostname` localhost 0
# Initialize the network hardware
/etc/ifconfig  ex0  `hostname`  -trailers  up # for Ethernet
/etc/slattach  /dev/ttyaf  19200 # Serial-Link to grumpy
/etc/ifconfig  sl0  doc  grumpy  -trailers  up
/etc/slattach  /dev/ttyae  19200 # Serial-Link to sneezy
/etc/ifconfig  sl1  doc  sneezy  -trailers  up
# other stuff for initializing the system follows...
```

The loopback interface is initialized first and then the Ethernet interface for *doc* is initialized, the name of the Ethernet interface being "`ex0`". The Serial-Link to ***grumpy*** is then allocated by the **/etc/slattach** command. The port to be used is `/dev/ttyaf` and the baud rate is set to `19200`, the maximum baud rate available. The interface just allocated is then configured by the **/etc/ifconfig** command. The interface name is "`sl0`," the connection is from ***doc*** to ***grumpy***, no "`trailer`" link-level encapsulation is supported, and the interface is marked as "`up`". The first **/etc/slattach** command sets up interface "`sl0`," the second, "`sl1`," and so on. The subsequent **/etc/ifconfig** command must use the interface name for that Serial-Link.

The /etc/uxrc File

The */etc/uxrc* file is used at system boot time to determine the modem control and hardware handshaking configurations of each port on the PCP16 board. Normally, ports used for Serial-Link connections are not set for modem control nor hardware handshaking. This is the default setting for all ports in the ICON/UXV release. If the */etc/uxrc* file has been changed from the default, use the ICON/UXV operating system reference manuals to learn how to adjust the file for your environment.

The /usr/lib/sendmail.cf File

The */usr/lib/sendmail.cf* file is used to determine how to send mail among hosts. It is somewhat cryptic and can be configured in different ways. The */usr/lib/sendmail.cf* file provided with ICON/UXV-NET software is usually sufficient. Further information can be obtained by referring to the appendices.

Reboot and Test

After modifying the files described above, the following steps must be taken:

- perform a system shutdown
- connect any yet unconnected cables
- reboot the system

The network can then be tested by attempting **rlogin**, **remsh**, **netstat**, and other network commands. The **ping** command is also useful for testing network configurations. If any problems are found after the above files have been configured, review the installation procedures outlined above and make sure the entries to each file are correct. If problems still persist, or you have questions concerning the configuration procedure, contact ICON Customer Service on the toll-free hot-line number: 1-800-444-ICON.

Network Setup 4

ICON/UXV provides support for the DARPA standard Internet protocols IP, ICMP, TCP, and UDP. These protocols may be used on top of a variety of hardware devices. Network services are split between the kernel (communication protocols) and user programs (user services such as TELNET and FTP). This section describes how to configure your system to use the Internet networking support.

All network interface drivers including the loopback interface, require that their host address(es) be defined at boot time. This is done with *ifconfig(8C)* commands included in the */etc/rc.local* file. Interfaces that are able to dynamically deduce the host part of an address may check that the host part of the address is correct. The manual page for each network interface describes the method used to establish a host's address. *Ifconfig(8)* can also be used to set options for the interface at boot time. Options are set independently for each interface, and apply to all packets sent using that interface. These options include disabling the use of the Address Resolution Protocol; this may be useful if a network is shared with hosts running software that does not yet provide this function. Alternatively, translations for such hosts may be set in advance or "published" by a ICON/UXV host by use of the *arp(8C)* command. Note that the use of trailer link-level is now negotiated between ICON/UXV hosts using ARP.

To use the pseudo terminals just configured, device entries must be created in the /dev directory. (These entries may already have been created on your system.) To create 32 pseudo terminals (plenty, unless you have a heavy network load) execute the following commands.

```
# cd /dev
# MAKEDEV pty0 pty1
```

More pseudo terminals may be made by specifying *pty2*, *pty3*, etc. The kernel normally includes support for 32 pseudo terminals unless the configuration file specifies a different number. Each pseudo terminal really consists of two files in /dev: a master and a slave. The master pseudo terminal file is named /dev/ptyp?, while the slave side is /dev/ttyp?. Pseudo terminals are also used by several programs not related to the network. In addition to creating the pseudo terminals, be sure to install them in the */etc/inittab* file (with an 'off' in the third field so no *getty* is started).

Local Subnetworks

NOTE: This section may be skipped on most systems.

In ICON/UXV the DARPA Internet support includes the notion of "subnetworks". This is a mechanism by which multiple local networks may appears as a single Internet network to off-site hosts. Subnetworks are useful because they allow a site to hide their local topology, requiring only a single route in external gateways; it also means that local network numbers may be locally administered. The standard describing this change in Internet addressing is RFC-950.

To set up local subnetworks one must first decide how the available address space (the Internet "host part" of the 32-bit address) is to be partitioned. Sites with a class A network number have a 24-bit address space with which to work, sites with a class B network number have a 16-bit address space, while sites with a class C network number have an 8-bit address space. To define local subnets you must steal some bits from the local host address space for use in extending the network portion of the Internet address. This reinterpretation of Internet addresses is done only for local networks; i.e. it is not visible to hosts off-site. For example, if your site has a class B network number, hosts on this network have an Internet address that contains the network number, 16 bits, and the host number, another 16 bits. To define 254 local subnets, each possessing at most 255 hosts, 8 bits may be taken from the local part. (The use of subnets 0 and all-1's, 255 in this example, is discouraged to avoid confusion about broadcast addresses.) These new network numbers are then constructed by concatenating the original 16-bit network number with the extra 8 bits containing the local subnetwork number.

The existence of local subnetworks is communicated to the system at the time a network interface is configured with the *netmask* option to the *ifconfig* program. A "network mask" is specified to define the portion of the Internet address that is to be considered the network part for that network. This mask normally contains the bits corresponding to the standard network part as well as the portion of the local part that has been assigned to subnets. If no mask is specified when the address is set, it will be set according to the class of the network. For example, at Berkeley (class B network 128.32) 8 bits of the local part have been reserved for defining subnetworks; consequently the /etc/rc.local file contains lines of the form

```
    /etc/ifconfig ex0 netmask 0xffffff00 128.32.1.7
```

This specifies that for interface "ex0", the upper 24 bits of the Internet address should be used in calculating network numbers (netmask 0xffffff00), and the interface's Internet address is "128.32.1.7" (host 7 on network 128.32.1). Hosts *m* on sub-network *n* of this network would then have addresses of the form "128.32.*n*.*m*"; for example, host 99 on network 129 would have an address "128.32.129.99". For hosts with multiple interfaces, the network mask should be set for each interface, although in practice only the mask of the first interface on each network is actually used.

Internet Broadcast Addresses

The address defined as the broadcast address for Internet networks according to RFC-919 is the address with a host part of all 1's. The address used by 4.2BSD was the address with a host part of 0. ICON/UXV uses the standard broadcast address (all 1's) by default, but allows the broadcast address to be set (with *ifconfig*) for each interface. This allows networks consisting of both 4.2BSD and ICON/UXV hosts to coexist. In the presence of subnets, the broadcast address uses the subnet field as for normal host addresses, with the remaining host part set to 1's (or 0's, on a network that has not yet been converted). ICON/UXV hosts recognize and accept packets sent to the logical-network broadcast address as well as those sent to the subnet broadcast address, and when using an all-1's broadcast, also recognize and receive packets sent to host 0 as a broadcast.

Routing

If your environment allows access to networks not directly attached to your host you will need to set up routing information to allow packets to be properly routed. Two schemes are supported by the system. The first scheme employs the routing table management daemon */etc/routed* to maintain the system routing tables. The routing daemon uses a variant of the Xerox Routing Information Protocol to maintain up to date routing tables in a cluster of local area networks. By using the */etc/gateways* file, the routing daemon can also be used to initialize static routes to distant networks (see the next section for further discussion). When the routing daemon is started up (usually from */etc/rc*) it reads */etc/gateways* if it exists and installs those routes defined there, then broadcasts on each local network to which the host is attached to find other instances of the routing daemon. If any responses are received, the routing daemons cooperate in maintaining a globally consistent view of routing in the local environment. This view can be extended to include remote sites also running the routing daemon by setting up suitable entries in */etc/gateways*; consult the *routed* manual page for a more thorough discussion.

The second approach is to define a default or wildcard route to a smart gateway and depend on the gateway to provide ICMP routing redirect information to dynamically create a routing data base. This is done by adding an entry of the form

```
     /etc/route add default smart-gateway 1
```

to */etc/rc*; see *routed(8C)* for more information. The default route will be used by the system as a "last resort" in routing packets to their destination. Assuming the gateway to which packets are directed is able to generate the proper routing redirect messages, the system will then add routing table entries based on the information supplied. This approach has certain advantages over the routing daemon, but is unsuitable in an environment where there are only bridges (i.e. pseudo gateways that, for instance, do not generate routing redirect messages). Further, if the smart gateway goes down there is no alternative, save manual alteration of the routing table entry, to maintaining service.

The system always listens, and processes, routing redirect information, so it is possible to combine both of the above facilities. For example, the routing table management process might be used to maintain up to date information about routes to geographically local networks, while employing the wildcard routing techniques for "distant" networks. The *netstat(1)* program may be used to display routing table contents as well as various routing oriented statistics. For example,

```
     # netstat  -r
```

will display the contents of the routing tables, while

```
     # netstat  -r  -s
```

will show the number of routing table entries dynamically created as a result of routing redirect messages, etc.

Network Servers

In ICON/UXV most of the server programs are started up by a "super server", the Internet daemon. The Internet daemon, */etc/inetd*, acts as a master server for programs specified in its configuration file, */etc/inetd.conf*, listening for service requests for these servers, and starting up the appropriate program whenever a request is received. The configuration file contains lines containing a service name (as found in */etc/services*), the type of socket the server expects (e.g. stream or dgram), the protocol to be used with the socket (as found in */etc/protocols*), whether to wait for each server to complete before starting up another, the user name as which the server should run, the server program's name, and at most five arguments to pass to the server program. Some trivial services are implemented internally in *inetd*, and their servers are listed as "internal." For example, an entry for the file transfer protocol server would appear as

```
ftp   stream   tcp   nowait   root   /etc/ftpd   ftpd
```

Consult the *inetd* manual page for more detail on the format of the configuration file and the operation of the Internet daemon.

Network Data Bases

Several data files are used by the network library routines and server programs. Most of these files are host independent and updated only rarely.

File	Manual reference	Use
/etc/hosts	*hosts*	host names
/etc/networks	*networks*	network names
/etc/services	*services*	list of known services
/etc/protocols	*protocols*	protocol names
/etc/hosts.equiv	*remshd*	list of "trusted" hosts
/etc/rc.local	*rc*	command script for starting servers
/etc/ftpusers	*ftpd*	list of "unwelcome" ftp users
/etc/hosts.lpd	*lpd*	list of hosts allowed to access printers
/etc/inetd.conf	*inetd*	list of servers started by *inetd*
/etc/gateways	*routed*	default router information

The files distributed are set up for ARPANET or other Internet hosts. Local networks and hosts should be added to describe the local configuration. Network numbers will have to be chosen for each Ethernet. For sites not connected to the Internet, these can be chosen more or less arbitrarily, otherwise the formal channels mentioned in Section 2 should be used for allocation of network numbers.

Regenerating /etc/hosts and /etc/networks

NOTE: The following information applies only to those system directly connected to the ARPA internet.

When using the host address routines that use the Internet name server, the file */etc/hosts* is only used for setting interface addresses and at other times that the server is not running, and therefore it need only contain addresses for local hosts. There is no equivalent service for network names yet. The full host and network name data bases are sometimes derived from a file retrieved from the Internet Network Information Center at SRI. To do this you should use the program /etc/gettable to retrieve the NIC host data base, and the program *htable(8)* to convert it to the format used by the libraries. You should change to the directory where you maintain your local additions to the host table and execute the following commands.

```
# /etc/gettable sri-nic.arpa
Connection to sri-nic.arpa opened.
Host table received.
Connection to sri-nic.arpa closed.
# /etc/htable hosts.txt
Warning, no localgateways file.
#
```

The *htable* program generates three files in the local directory: *hosts*, *networks* and *gateways*. If a file "localhosts" is present in the working directory its contents are first copied to the output file. Similarly, a "localnetworks" file may be prepended to the output created by *htable*, and "localgateways" will be prepended to *gateways*. It is usually wise to run *diff*(1) on the new host and network data bases before installing them in /etc. If you are using the name server for the host name and address mapping, you only need to install *networks* and a small copy of *hosts* describing your local machines. The full host table in this case might be placed somewhere else for reference by users. The gateways file may be installed in */etc/gateways* if you use *routed* for local routing and wish to have static external routes installed when *routed* is started. This procedure is essentially obsolete, however, except for individual hosts that are on the Arpanet or Milnet and do not forward packets from a local network. Other situations require the use of an EGP server.

If you are connected to the DARPA Internet, it is highly recommended that you use the name server for your host name and address mapping, as this provides access to a much larger set of hosts than are provided in the host table. Many large organization on the network, currently have only a small percentage of their hosts listed in the host table retrieved from NIC.

/etc/hosts.equiv

The remote login and shell servers use an authentication scheme based on trusted hosts. The *hosts.equiv* file contains a list of hosts that are considered trusted and, under a single administrative control. When a user contacts a remote login or shell server requesting service, the client process passes the user's name and the official name of the host on which the client is located. In the simple case, if the host's name is located in *hosts.equiv* and the user has an account on the server's machine, then service is rendered (i.e. the user is allowed to log in, or the command is executed). Users may expand this "equivalence" of machines by installing a

.rhosts file in their login directory. The root login is handled specially, bypassing the *hosts.equiv* file, and using only the */.rhosts* file.

Thus, to create a class of equivalent machines, the *hosts.equiv* file should contain the *official* names for those machines. If you are running the name server, you may omit the domain part of the host name for machines in your local domain. For example, several machines on our local network are considered trusted, so the *hosts.equiv* file is of the form:

```
dopey
doc
sneezy
happy
grumpy
sleepy
```

/etc/rc.local

Most network servers are automatically started up at boot time by the command file /etc/rc (if they are installed in their presumed locations) or by the Internet daemon (see above). These include the following:

Program	Server	Started by
/etc/remshd	shell server	inetd
/etc/rexecd	exec server	inetd
/etc/rlogind	login server	inetd
/etc/telnetd	TELNET server	inetd
/etc/ftpd	FTP server	inetd
/etc/fingerd	Finger server	inetd
/etc/tftpd	TFTP server	inetd
/etc/rwhod	system status daemon	/etc/rc
/etc/syslogd	error logging server	/etc/rc
/usr/lib/sendmail	SMTP server	/etc/rc
/etc/routed	routing table management daemon	/etc/rc

Consult the manual pages and accompanying documentation (particularly for sendmail) for details about their operation.

To have other network servers started up as well, the appropriate line should be added to the Internet daemon's configuration file */etc/inetd.conf*, or commands of the following sort should be placed in the file */etc/rc*.

```
if [ -f /etc/routed ]; then
     /etc/routed & echo ' routed\c'
fi
```

/etc/ftpusers

The FTP server included in the system provides support for an anonymous FTP account. Because of the inherent security problems with such a facility you should read this section carefully if you consider providing such a service.

An anonymous account is enabled by creating a user *ftp*. When a client uses the anonymous account a *chroot*(2) system call is performed by the server to restrict the client from moving outside that part of the file system where the user ftp home directory is located. Because a *chroot* call is used, certain programs and files used by the server process must be placed in the ftp home directory. Further, one must be sure that all directories and executable images are unwritable. The following directory setup is recommended. (Note: The *csh* shell is used in the following examples.)

```
# cd ~ftp
# chmod 555 .; chown ftp .; chgrp ftp .
# mkdir bin etc pub
# chown root bin etc
# chmod 555 bin etc
# chown ftp pub
# chmod 777 pub
# cd bin
# cp /bin/sh /bin/ls .
# chmod 111 sh ls
# cd ../etc
# cp /etc/passwd /etc/group .
# chmod 444 passwd group
```

When local users wish to place files in the anonymous area, they must be placed in a subdirectory. In the setup here, the directory *~ftp/pub* is used.

Another issue to consider is the copy of */etc/passwd* placed here. It may be copied by users who use the anonymous account. They may then try to break the passwords of users on your machine for further access. A good choice of users to include in this copy might be root, daemon, uucp, and the ftp user. All passwords here should probably be "*".

Aside from the problems of directory modes and such, the ftp server may provide a loophole for interlopers if certain user accounts are allowed. The file */etc/ftpusers* is checked on each connection. If the requested user name is located in the file, the request for service is denied. This file normally has the following names:

```
uucp
root
```

Accounts with nonstandard shells should be listed in this file. Accounts without passwords need not be listed in this file, the ftp server will not service these users.

Appendices

The following Appendices contain manual pages pertaining to commands and utilities contained in the ICON/UXV-NET networking package, a document on SENDMAIL, the Internetwork Mail Sender, and documents on the Internet Protocols, SENDMAIL Installation and Operation, and 4.3BSD Notes on Network Implementation.

Appendix A – Manual Pages

The following appendix contains manual pages pertaining to the ICON/UXV-NET commands and utilities that make up the networking package described in the previous sections. The pages are divided into four classes:

Section	Section Number
User Commands	1 and 1C
Maintenance Commands	1M
System Calls	2
Subroutines	3
Network Functions	3N
File Formats	4
Miscellaneous	5
Networking Protocol Families	7N
Networking Protocols	7P

The following is a list of the commands and utilities that are contained in each section.

User Commands (1 and 1C)

ftp	ARPANET file transfer program
netstat	show network status
newaliases	rebuild the data base for the mail aliases file
rcp	remote file copy
rdist	remote file distribution program
rlogin	remote login
remsh	remote shell
ruptime	show host status of local machines
rwho	who's logged in on local machines
talk	talk to another user
telnet	user interface to the TELNET protocol
tftp	trivial file transfer program

Maintenance Commands (1M)

ftpd	DARPA Internet File Transfer Protocol server
gettable	get NIC format host tables from a host
htable	convert NIC standard format host tables
ifconfig	configure network interface parameters
inetd	internet "super-server"
ping	send ICMP ECHO_REQUEST packets to network hosts
rexecd	remote execution server
rlogind	remote login server
route	manually manipulate the routing tables
routed	network routing daemon
remshd	remote shell server
rwhod	system status server
sendmail	send mail over the internet

Maintenance Commands (1M) (Continued)

slattach	attach serial lines as network interfaces
syslogd	log systems messages
talkd	remote user communication server
telnetd	DARPA TELNET protocol server
tftpd	DARPA Trivial File Transfer Protocol server
trpt	transliterate protocol trace

System Calls (2)

accept	accept a connection on a socket
bind	bind a name to a socket
connect	initiate a connection on a socket
fchmod	change mode of file
fchown	change owner and group of a file
gethostid	get unique identifier of current host
gethostname	get name of current host
getpeername	get name of connected peer
getsockname	get socket name
getsockopt	get options on sockets
gettimeofday	get date and time
listen	listen for connections on a socket
readv	read input
recv	receive a message from a socket
recvfrom	receive a message from a socket
recvmsg	receive a message from a socket
select	synchronous I/O multiplexing
send	send a message from a socket
sendto	send a message from a socket
sendmsg	send a message from a socket
sethostid	set unique identifier of current host
sethostname	set name of current host
setsockopt	set options on sockets
settimeofday	set date and time
shutdown	shut down part of a full-duplex connection
socket	create an endpoint for communication
socketpair	create a pair of connected sockets
vfork	spawn new process in a virtual memory efficient way
writev	write output

Subroutines (3)

rcmd	routines for returning a stream to a remote command
rexec	return stream to a remote command
rresvport	routines for returning a stream to a remote command
ruserok	routines for returning a stream to a remote command

Network Functions (3N)

endhostent	end network host entry
endnetent	end network entry
endprotoent	end protocol entry
endtservent	end service entry
gethostbyaddr	get network host entry by address
gethostbyname	get network host entry by name
gethostent	get network host entry
getnetbyaddr	get network entry by address
getnetbyname	get network entry by name
getnetent	get network entry
getprotoent	get protocol entry
getprotoentbyname	get protocol entry by name
getprotoentbynumber	get protocol entry by number
getservbyname	get service entry by name
getservbyport	get service entry by port
getservent	get service entry
herror	get network host entry error
htonl	convert values between host and network byte order
htons	convert values between host and network byte order
inet_addr	Internet address manipulation routine
inet_lnaof	Internet address manipulation routine
inet_makeaddr	Internet address manipulation routine
inet_network	Internet address manipulation routine
inet_netof	Internet address manipulation routine
inet_ntoa	Internet address manipulation routine
ntohl	convert values between host and network byte order
ntohs	convert values between host and network byte order
sethostent	set network host entry
setnetent	set network entry
setprotoent	set protocol entry
setservent	set service entry

File Formats (4)

aliases	aliases file for sendmail
hosts	host name data base
networks	network name data base
protocols	protocol name data base
services	service name data base

Miscellaneous (5)

hostname	host name resolution description
mailaddr	mail addressing description
resolver	resolver configuration file

Networking Protocols (7N)

networking intro	introductin to networking facilities

Networking Protocol Families (7P)

arp	Address Resolution Protocol
icmp	Internet Control Message Protocol
ip	Internet Protocol
tcp	Internet Transmission Control Protocol
udp	Internet User Datagram Protocol

NAME

ftp - ARPANET file transfer program

SYNOPSIS

ftp [**-v**] [**-d**] [**-i**] [**-n**] [**-g**] [**host**]

DESCRIPTION

Ftp is the user interface to the ARPANET standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site.

The client host with which *ftp* is to communicate may be specified on the command line. If this is done, *ftp* will immediately attempt to establish a connection to an FTP server on that host; otherwise, *ftp* will enter its command interpreter and await instructions from the user. When *ftp* is awaiting commands from the user the prompt ''ftp>'' is provided to the user. The following commands are recognized by *ftp*:

! [*command* [*args*]]
: Invoke an interactive shell on the local machine. If there are arguments, the first is taken to be a command to execute directly, with the rest of the arguments as its arguments.

$ *macro-name* [*args*]
: Execute the macro *macro-name* that was defined with the **macdef** command. Arguments are passed to the macro unglobbed.

account [*passwd*]
: Supply a supplemental password required by a remote system for access to resources once a login has been successfully completed. If no argument is included, the user will be prompted for an account password in a non-echoing input mode.

append *local-file* [*remote-file*]
: Append a local file to a file on the remote machine. If *remote-file* is left unspecified, the local file name is used in naming the remote file after being altered by any *ntrans* or *nmap* setting. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

ascii Set the file transfer *type* to network ASCII. This is the default type.

bell Arrange that a bell be sounded after each file transfer command is completed.

binary Set the file transfer *type* to support binary image transfer.

bye Terminate the FTP session with the remote server and exit *ftp*. An end of file will also terminate the session and exit.

case Toggle remote computer file name case mapping during **mget** commands. When **case** is on (default is off), remote computer file names with all letters in upper case are written in the local directory with the letters mapped to lower case.

cd *remote-directory*
: Change the working directory on the remote machine to *remote-directory*.

cdup Change the remote machine working directory to the parent of the current remote machine working directory.

close Terminate the FTP session with the remote server, and return to the command interpreter. Any defined macros are erased.

cr Toggle carriage return stripping during ascii type file retrieval. Records are denoted by a carriage return/linefeed sequence during ascii type file transfer. When **cr** is on (the default), carriage returns are stripped from this sequence to conform with the UNIX single linefeed record delimiter. Records on non-UNIX remote systems may contain single linefeeds; when an ascii type transfer is made, these linefeeds may be distinguished from a record delimiter only when **cr** is off.

delete *remote-file*
Delete the file *remote-file* on the remote machine.

debug [*debug-value*]
Toggle debugging mode. If an optional *debug-value* is specified it is used to set the debugging level. When debugging is on, *ftp* prints each command sent to the remote machine, preceded by the string "-->".

dir [*remote-directory*] [*local-file*]
Print a listing of the directory contents in the directory, *remote-directory*, and, optionally, placing the output in *local-file*. If no directory is specified, the current working directory on the remote machine is used. If no local file is specified, or *local-file* is **-**, output comes to the terminal.

disconnect
A synonym for **close**.

form *format*
Set the file transfer *form* to *format*. The default format is "file".

get *remote-file* [*local-file*]
Retrieve the *remote-file* and store it on the local machine. If the local file name is not specified, it is given the same name it has on the remote machine, subject to alteration by the current *case*, *ntrans*, and *nmap* settings. The current settings for *type*, *form*, *mode*, and *structure* are used while transferring the file.

glob Toggle filename expansion for **mdelete**, **mget** and **mput**. If globbing is turned off with **glob**, the file name arguments are taken literally and not expanded. Globbing for **mput** is done as in **csh**(1). For **mdelete** and **mget**, each remote file name is expanded separately on the remote machine and the lists are not merged. Expansion of a directory name is likely to be different from expansion of the name of an ordinary file: the exact result depends on the foreign operating system and ftp server, and can be previewed by doing '**mls** *remote-files* **-**'. Note: **mget** and **mput** are not meant to transfer entire directory subtrees of files. That can be done by transferring a **tar**(1) archive of the subtree (in binary mode).

hash Toggle hash-sign ("#") printing for each data block transferred. The size of a data block is 1024 bytes.

help [*command*]
Print an informative message about the meaning of *command*. If no argument is given, *ftp* prints a list of the known commands.

lcd [*directory*]
Change the working directory on the local machine. If no *directory* is specified, the user's home directory is used.

ls [*remote-directory*] [*local-file*]
Print an abbreviated listing of the contents of a directory on the remote machine. If *remote-directory* is left unspecified, the current working directory is used. If no local

file is specified, or if *local-file* is **-**, the output is sent to the terminal.

macdef *macro-name*
: Define a macro. Subsequent lines are stored as the macro *macro-name*; a null line (consecutive newline characters in a file or carriage returns from the terminal) terminates macro input mode. There is a limit of 16 macros and 4096 total characters in all defined macros. Macros remain defined until a **close** command is executed. The macro processor interprets '$' and '\' as special characters. A '$' followed by a number (or numbers) is replaced by the corresponding argument on the macro invocation command line. A '$' followed by an 'i' signals that macro processor that the executing macro is to be looped. On the first pass '$i' is replaced by the first argument on the macro invocation command line, on the second pass it is replaced by the second argument, and so on. A '\' followed by any character is replaced by that character. Use the '\' to prevent special treatment of the '$'.

mdelete [*remote-files*]
: Delete the *remote-files* on the remote machine.

mdir *remote-files local-file*
: Like **dir**, except multiple remote files may be specified. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **mdir** output.

mget *remote-files*
: Expand the *remote-files* on the remote machine and do a **get** for each file name thus produced. See **glob** for details on the filename expansion. Resulting file names will then be processed according to *case*, *ntrans*, and *nmap* settings. Files are transferred into the local working directory, which can be changed with '**lcd** directory'; new local directories can be created with '**!** mkdir directory'.

mkdir *directory-name*
: Make a directory on the remote machine.

mls *remote-files local-file*
: Like **ls**, except multiple remote files may be specified. If interactive prompting is on, *ftp* will prompt the user to verify that the last argument is indeed the target local file for receiving **mls** output.

mode [*mode-name*]
: Set the file transfer *mode* to *mode-name*. The default mode is ''stream'' mode.

mput *local-files*
: Expand wild cards in the list of local files given as arguments and do a **put** for each file in the resulting list. See **glob** for details of filename expansion. Resulting file names will then be processed according to *ntrans* and *nmap* settings.

nmap [*inpattern outpattern*]
: Set or unset the filename mapping mechanism. If no arguments are specified, the filename mapping mechanism is unset. If arguments are specified, remote filenames are mapped during **mput** commands and **put** commands issued without a specified remote target filename. If arguments are specified, local filenames are mapped during **mget** commands and **get** commands issued without a specified local target filename. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices. The mapping follows the pattern set by *inpattern* and *outpattern*. *Inpattern* is a template for incoming filenames (which may have already been processed according to the **ntrans** and **case** settings). Variable

templating is accomplished by including the sequences '$1', '$2', ..., '$9' in *inpattern*. Use '\' to prevent this special treatment of the '$' character. All other characters are treated literally, and are used to determine the **nmap** *inpattern* variable values. For exmaple, given *inpattern* $1.$2 and the remote file name "mydata.data", $1 would have the value "mydata", and $2 would have the value "data". The *outpattern* determines the resulting mapped filename. The sequences '$1', '$2', ..., '$9' are replaced by any value resulting from the *inpattern* template. The sequence '$0' is replace by the original filename. Additionally, the sequence '[*seq1*,*seq2*]' is replaced by *seq1* if *seq1* is not a null string; otherwise it is replaced by *seq2*. For example, the command "nmap $1.$2.$3 [$1,$2].[$2,file]" would yield the output filename "myfile.data" for input filenames "myfile.data" and "myfile.data.old", "myfile.file" for the input filename "myfile", and "myfile.myfile" for the input filename ".myfile". Spaces may be included in *outpattern*, as in the example: nmap $1 |sed "s/ *$//" > $1 . Use the '\' character to prevent special treatment of the '$', '[', ']', and ',' characters.

ntrans [*inchars* [*outchars*]]

Set or unset the filename character translation mechanism. If no arguments are specified, the filename character translation mechanism is unset. If arguments are specified, characters in remote filenames are translated during **mput** commands and **put** commands issued without a specified remote target filename. If arguments are specified, characters in local filenames are translated during **mget** commands and **get** commands issued without a specified local target filename. This command is useful when connecting to a non-UNIX remote computer with different file naming conventions or practices. Characters in a filename matching a character in *inchars* are replaced with the corresponding character in *outchars*. If the character's position in *inchars* is longer than the length of *outchars*, the character is deleted from the file name.

open *host* [*port*]

Establish a connection to the specified *host* FTP server. An optional port number may be supplied, in which case, *ftp* will attempt to contact an FTP server at that port. If the *auto-login* option is on (default), *ftp* will also attempt to automatically log the user in to the FTP server (see below).

prompt

Toggle interactive prompting. Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files. If prompting is turned off (default is on), any **mget** or **mput** will transfer all files, and any **mdelete** will delete all files.

proxy *ftp-command*

Execute an ftp command on a secondary control connection. This command allows simultaneous connection to two remote ftp servers for transferring files between the two servers. The first **proxy** command should be an **open**, to establish the secondary control connection. Enter the command "proxy ?" to see other ftp commands executable on the secondary connection. The following commands behave differently when prefaced by **proxy**: **open** will not define new macros during the auto-login process, **close** will not erase existing macro definitions, **get** and **mget** transfer files from the host on the primary control connection to the host on the secondary control connection, and **put**, **mput**, and **append** transfer files from the host on the secondary control connection to the host on the primary control connection. Third party file transfers depend upon support of the ftp protocol PASV command by the server on the secondary control connection.

put *local-file* [*remote-file*]
: Store a local file on the remote machine. If *remote-file* is left unspecified, the local file name is used after processing according to any *ntrans* or *nmap* settings in naming the remote file. File transfer uses the current settings for *type*, *format*, *mode*, and *structure*.

pwd Print the name of the current working directory on the remote machine.

quit A synonym for **bye**.

quote *arg1 arg2 ...*
: The arguments specified are sent, verbatim, to the remote FTP server.

recv *remote-file* [*local-file*]
: A synonym for get.

remotehelp [*command-name*]
: Request help from the remote FTP server. If a *command-name* is specified it is supplied to the server as well.

rename [*from*] [*to*]
: Rename the file *from* on the remote machine, to the file *to*.

reset Clear reply queue. This command re-synchronizes command/reply sequencing with the remote ftp server. Resynchronization may be neccesary following a violation of the ftp protocol by the remote server.

rmdir *directory-name*
: Delete a directory on the remote machine.

runique
: Toggle storing of files on the local system with unique filenames. If a file already exists with a name equal to the target local filename for a **get** or **mget** command, a ".1" is appended to the name. If the resulting name matches another existing file, a ".2" is appended to the original name. If this process continues up to ".99", an error message is printed, and the transfer does not take place. The generated unique filename will be reported. Note that **runique** will not affect local files generated from a shell command (see below). The default value is off.

send *local-file* [*remote-file*]
: A synonym for put.

sendport
: Toggle the use of PORT commands. By default, *ftp* will attempt to use a PORT command when establishing a connection for each data transfer. The use of PORT commands can prevent delays when performing multiple file transfers. If the PORT command fails, *ftp* will use the default data port. When the use of PORT commands is disabled, no attempt will be made to use PORT commands for each data transfer. This is useful for certain FTP implementations which do ignore PORT commands but, incorrectly, indicate they've been accepted.

status Show the current status of *ftp*.

struct [*struct-name*]
: Set the file transfer *structure* to *struct-name*. By default ''stream'' structure is used.

sunique
: Toggle storing of files on remote machine under unique file names. Remote ftp server must support ftp protocol STOU command for successful completion. The remote

server will report unique name. Default value is off.

tenex Set the file transfer type to that needed to talk to TENEX machines.

trace Toggle packet tracing.

type [*type-name*]
Set the file transfer *type* to *type-name*. If no type is specified, the current type is printed. The default type is network ASCII.

user *user-name* [*password*] [*account*]
Identify yourself to the remote FTP server. If the password is not specified and the server requires it, *ftp* will prompt the user for it (after disabling local echo). If an account field is not specified, and the FTP server requires it, the user will be prompted for it. If an account field is specified, an account command will be relayed to the remote server after the login sequence is completed if the remote server did not require it for logging in. Unless *ftp* is invoked with ''auto-login'' disabled, this process is done automatically on initial connection to the FTP server.

verbose
Toggle verbose mode. In verbose mode, all responses from the FTP server are displayed to the user. In addition, if verbose is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, verbose is on.

? [*command*]
A synonym for help.

Command arguments which have embedded spaces may be quoted with quote (") marks.

ABORTING A FILE TRANSFER

To abort a file transfer, use the terminal interrupt key (usually Ctrl-C). Sending transfers will be immediately halted. Receiving transfers will be halted by sending a ftp protocol ABOR command to the remote server, and discarding any further data received. The speed at which this is accomplished depends upon the remote server's support for ABOR processing. If the remote server does not support the ABOR command, an "ftp>" prompt will not appear until the remote server has completed sending the requested file.

The terminal interrupt key sequence will be ignored when *ftp* has completed any local processing and is awaiting a reply from the remote server. A long delay in this mode may result from the ABOR processing described above, or from unexpected behavior by the remote server, including violations of the ftp protocol. If the delay results from unexpected remote server behavior, the local *ftp* program must be killed by hand.

FILE NAMING CONVENTIONS

Files specified as arguments to *ftp* commands are processed according to the following rules.

1) If the file name ''-'' is specified, the **stdin** (for reading) or **stdout** (for writing) is used.

2) If the first character of the file name is ''|'', the remainder of the argument is interpreted as a shell command. *Ftp* then forks a shell, using *popen*(3) with the argument supplied, and reads (writes) from the stdout (stdin). If the shell command includes spaces, the argument must be quoted; e.g. ''"| ls -lt"''. A particularly useful example of this mechanism is: ''dir |more''.

3) Failing the above checks, if ''globbing'' is enabled, local file names are expanded

according to the rules used in the *csh*(1); c.f. the *glob* command. If the *ftp* command expects a single local file (.e.g. **put**), only the first filename generated by the "globbing" operation is used.

4) For **mget** commands and **get** commands with unspecified local file names, the local filename is the remote filename, which may be altered by a **case**, **ntrans**, or **nmap** setting. The resulting filename may then be altered if **runique** is on.

5) For **mput** commands and **put** commands with unspecified remote file names, the remote filename is the local filename, which may be altered by a **ntrans** or **nmap** setting. The resulting filename may then be altered by the remote server if **sunique** is on.

FILE TRANSFER PARAMETERS

The FTP specification specifies many parameters which may affect a file transfer. The *type* may be one of ''ascii'', ''image'' (binary), ''ebcdic'', and ''local byte size'' (for PDP-10's and PDP-20's mostly). *Ftp* supports the ascii and image types of file transfer, plus local byte size 8 for **tenex** mode transfers.

Ftp supports only the default values for the remaining file transfer parameters: *mode*, *form*, and *struct*.

OPTIONS

Options may be specified at the command line, or to the command interpreter.

The **-v** (verbose on) option forces *ftp* to show all responses from the remote server, as well as report on data transfer statistics.

The **-n** option restrains *ftp* from attempting ''auto-login'' upon initial connection. If auto-login is enabled, *ftp* will check the *.netrc* (see below) file in the user's home directory for an entry describing an account on the remote machine. If no entry exists, *ftp* will prompt for the remote machine login name (default is the user identity on the local machine), and, if necessary, prompt for a password and an account with which to login.

The **-i** option turns off interactive prompting during multiple file transfers.

The **-d** option enables debugging.

The **-g** option disables file name globbing.

THE .netrc FILE

The .netrc file contains login and initialization information used by the auto-login process. It resides in the user's home directory. The following tokens are recognized; they may be separated by spaces, tabs, or new-lines:

machine *name*

Identify a remote machine name. The auto-login process searches the .netrc file for a **machine** token that matches the remote machine specified on the *ftp* command line or as an **open** command argument. Once a match is made, the subsequent .netrc tokens are processed, stopping when the end of file is reached or another **machine** token is encountered.

login *name*

Identify a user on the remote machine. If this token is present, the auto-login process will initiate a login using the specified name.

password *string*
Supply a password. If this token is present, the auto-login process will supply the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the .netrc file, *ftp* will abort the auto-login process if the .netrc is readable by anyone besides the user.

account *string*
Supply an additional account password. If this token is present, the auto-login process will supply the specified string if the remote server requires an additional account password, or the auto-login process will initiate an ACCT command if it does not.

macdef *name*
Define a macro. This token functions like the *ftp* **macdef** command functions. A macro is defined with the specified name; its contents begin with the next .netrc line and continue until a null line (consecutive new-line characters) is encountered. If a macro named *init* is defined, it is automatically executed as the last step in the auto-login process.

BUGS

Correct execution of many commands depends upon proper behavior by the remote server.

An error in the treatment of carriage returns in the 4.2BSD UNIX ascii-mode transfer code has been corrected. This correction may result in incorrect transfers of binary files to and from 4.2BSD servers using the ascii type. Avoid this problem by using the binary image type.

NAME

netstat - show network status

SYNOPSIS

netstat [**-Aan**] [**-f** *address_family*] [*system*] [*core*]
netstat [**-himnrs**] [**-f** *address_family*] [*system*] [*core*]
netstat [**-n**] [**-I** *interface*] *interval* [*system*] [*core*]

DESCRIPTION

The *netstat* command symbolically displays the contents of various network-related data structures. There are a number of output formats, depending on the options for the information presented. The first form of the command displays a list of active sockets for each protocol. The second form presents the contents of one of the other network data structures according to the option selected. Using the third form, with an *interval* specified, *netstat* will continuously display the information regarding packet traffic on the configured network interfaces.

The options have the following meaning:

-A　With the default display, show the address of any protocol control blocks associated with sockets; used for debugging.

-a　With the default display, show the state of all sockets; normally sockets used by server processes are not shown.

-h　Show the state of the IMP host table.

-i　Show the state of interfaces which have been auto-configured (interfaces statically configured into a system, but not located at boot time are not shown).

-I *interface*
　Show information only about this interface; used with an *interval* as described below.

-m　Show statistics recorded by the memory management routines (the network manages a private pool of memory buffers).

-n　Show network addresses as numbers (normally *netstat* interprets addresses and attempts to display them symbolically). This option may be used with any of the display formats.

-s　Show per-protocol statistics.

-r　Show the routing tables. When **-s** is also present, show routing statistics instead.

-f *address_family*
　Limit statistics or address control block reports to those of the specified *address family*. The following address families are recognized: *inet*, for **AF_INET**, *ns*, for **AF_NS**, and *unix*, for **AF_UNIX**.

The arguments, *system* and *core* allow substitutes for the defaults ''/vmunix'' and ''/dev/kmem''.

The default display, for active sockets, shows the local and remote addresses, send and receive queue sizes (in bytes), protocol, and the internal state of the protocol. Address formats are of the form ''host.port'' or ''network.port'' if a socket's address specifies a network but no specific host address. When known the host and network addresses are displayed symbolically according to the data bases */etc/hosts* and */etc/networks*, respectively. If a symbolic name for an address is unknown, or if the **-n** option is specified, the address is printed numerically, according to the address family. For more information regarding the Internet ''dot format,'' refer to *inet*(3N). Unspecified, or ''wildcard'', addresses and ports appear as ''*''.

The interface display provides a table of cumulative statistics regarding packets transferred, errors, and collisions. The network addresses of the interface and the maximum transmission unit (''mtu'') are also displayed.

The routing table display indicates the available routes and their status. Each route consists of a destination host or network and a gateway to use in forwarding packets. The flags field shows the state of the route (''U'' if ''up''), whether the route is to a gateway (''G''), and whether the route was created dynamically by a redirect (''D''). Direct routes are created for each interface attached to the local host; the gateway field for such entries shows the address of the outgoing interface. The refcnt field gives the current number of active uses of the route. Connection oriented protocols normally hold on to a single route for the duration of a connection while connectionless protocols obtain a route while sending to the same destination. The use field provides a count of the number of packets sent using that route. The interface entry indicates the network interface utilized for the route.

When *netstat* is invoked with an *interval* argument, it displays a running count of statistics related to network interfaces. This display consists of a column for the primary interface (the first interface found during autoconfiguration) and a column summarizing information for all interfaces. The primary interface may be replaced with another interface with the *-I* option. The first line of each screen of information contains a summary since the system was last rebooted. Subsequent lines of output show values accumulated over the preceding interval.

SEE ALSO

hosts(4), networks(4), protocols(4), services(4), trpt(1M)

BUGS

The notion of errors is ill-defined. Collisions mean something else for the IMP.

NAME

newaliases - rebuild the data base for the mail aliases file

SYNOPSIS

newaliases

DESCRIPTION

Newaliases rebuilds the random access data base for the mail aliases file */usr/lib/aliases*. It must be run each time */usr/lib/aliases* is changed in order for the change to take effect.

SEE ALSO

aliases(4), sendmail(1M)

NAME

rcp - remote file copy

SYNOPSIS

rcp [**-p**] file1 file2
rcp [**-p**] [**-r**] file ... directory

DESCRIPTION

Rcp copies files between machines. Each *file* or *directory* argument is either a remote file name of the form "rhost:path", or a local file name (containing no ':' characters, or a '/' before any ':'s).

If the **-r** option is specified and any of the source files are directories, *rcp* copies each subtree rooted at that name; in this case the destination must be a directory.

By default, the mode and owner of *file2* are preserved if it already existed; otherwise the mode of the source file modified by the *umask*(2) on the destination host is used. The **-p** option causes *rcp* to attempt to preserve (duplicate) in its copies the modification times and modes of the source files, ignoring the *umask*.

If *path* is not a full path name, it is interpreted relative to your login directory on *rhost*. A *path* on a remote host may be quoted (using \, ", or ') so that the metacharacters are interpreted remotely.

Rcp does not prompt for passwords; your current local user name must exist on *rhost* and allow remote command execution via *remsh*(1C).

Rcp handles third party copies, where neither source nor target files are on the current machine. Hostnames may also take the form "rname@rhost" to use *rname* rather than the current user name on the remote host. The destination hostname may also take the form "rhost.rname" to support destination machines that are running 4.2BSD versions of *rcp*.

SEE ALSO

cp(1), ftp(1C), remsh(1C), rlogin(1C)

BUGS

Doesn't detect all cases where the target of a copy might be a file in cases where only a directory should be legal.
Is confused by any output generated by commands in a .login, .profile, or .cshrc file on the remote host.

NAME

rdist - remote file distribution program

SYNOPSIS

rdist [-nqbRhivwy] [-f distfile] [-d var=value] [-m host] [name ...]

rdist [-nqbRhivwy] -c name ... [login@]host[:dest]

DESCRIPTION

Rdist is a program to maintain identical copies of files over multiple hosts. It preserves the owner, group, mode, and mtime of files if possible and can update programs that are executing. *Rdist* reads commands from *distfile* to direct the updating of files and/or directories. If *distfile* is '-', the standard input is used. If no **-f** option is present, the program looks first for 'distfile', then 'Distfile' to use as the input. If no names are specified on the command line, *rdist* will update all of the files and directories listed in *distfile*. Otherwise, the argument is taken to be the name of a file to be updated or the label of a command to execute. If label and file names conflict, it is assumed to be a label. These may be used together to update specific files using specific commands.

The **-c** option forces *rdist* to interpret the remaining arguments as a small *distfile*. The equivalent distfile is as follows.

```
( name ... ) -> [login@]host
        install  [dest] ;
```

Other options:

-d Define *var* to have *value*. The **-d** option is used to define or override variable definitions in the *distfile*. *Value* can be the empty string, one name, or a list of names surrounded by parentheses and separated by tabs and/or spaces.

-m Limit which machines are to be updated. Multiple **-m** arguments can be given to limit updates to a subset of the hosts listed the *distfile*.

-n Print the commands without executing them. This option is useful for debugging *distfile*.

-q Quiet mode. Files that are being modified are normally printed on standard output. The **-q** option suppresses this.

-R Remove extraneous files. If a directory is being updated, any files that exist on the remote host that do not exist in the master directory are removed. This is useful for maintaining truely identical copies of directories.

-h Follow symbolic links. Copy the file that the link points to rather than the link itself.

-i Ignore unresolved links. *Rdist* will normally try to maintain the link structure of files being transfered and warn the user if all the links cannot be found.

-v Verify that the files are up to date on all the hosts. Any files that are out of date will be displayed but no files will be changed nor any mail sent.

-w Whole mode. The whole file name is appended to the destination directory name. Normally, only the last component of a name is used when renaming files. This will preserve the directory structure of the files being copied instead of flattening the directory structure. For example, renaming a list of files such as (dir1/f1 dir2/f2) to dir3 would create files dir3/dir1/f1 and dir3/dir2/f2 instead of dir3/f1 and dir3/f2.

-y Younger mode. Files are normally updated if their *mtime* and *size* (see *stat*(2)) disagree. The -y option causes *rdist* not to update files that are younger than the master copy. This can be used to prevent newer copies on other hosts from being replaced. A warning message is printed for files which are newer than the master copy.

-b Binary comparison. Perform a binary comparison and update files if they differ rather than comparing dates and sizes.

Distfile contains a sequence of entries that specify the files to be copied, the destination hosts, and what operations to perform to do the updating. Each entry has one of the following formats.

```
<variable name> '=' <name list>
[ label: ] <source list> '->' <destination list> <command list>
[ label: ] <source list> '::' <time_stamp file> <command list>
```

The first format is used for defining variables. The second format is used for distributing files to other hosts. The third format is used for making lists of files that have been changed since some given date. The *source list* specifies a list of files and/or directories on the local host which are to be used as the master copy for distribution. The *destination list* is the list of hosts to which these files are to be copied. Each file in the source list is added to a list of changes if the file is out of date on the host which is being updated (second format) or the file is newer than the time stamp file (third format).

Labels are optional. They are used to identify a command for partial updates.

Newlines, tabs, and blanks are only used as separators and are otherwise ignored. Comments begin with '#' and end with a newline.

Variables to be expanded begin with '$' followed by one character or a name enclosed in curly braces (see the examples at the end).

The source and destination lists have the following format:

```
        <name>
or
        '(' <zero or more names separated by white-space> ')'
```

The shell meta-characters '[', ']', '{', '}', '*', and '?' are recognized and expanded (on the local host only) in the same way as *csh*(1). They can be escaped with a backslash. The '~' character is also expanded in the same way as *csh* but is expanded separately on the local and destination hosts. When the -w option is used with a file name that begins with '~', everything except the home directory is appended to the destination name. File names which do not begin with '/' or '~' use the destination user's home directory as the root directory for the rest of the file name.

The command list consists of zero or more commands of the following format.

```
'install' <options> opt_dest_name ';'
'notify' <name list>';'
'except' <name list>';'
```

```
'except_pat'     <pattern list>';'
'special'        <name list>string ';'
```

The *install* command is used to copy out of date files and/or directories. Each source file is copied to each host in the destination list. Directories are recursively copied in the same way. *Opt_dest_name* is an optional parameter to rename files. If no *install* command appears in the command list or the destination name is not specified, the source file name is used. Directories in the path name will be created if they do not exist on the remote host. To help prevent disasters, a non-empty directory on a target host will never be replaced with a regular file or a symbolic link. However, under the '-R' option a non-empty directory will be removed if the corresponding filename is completely absent on the master host. The *options* are '-R', '-h', '-i', '-v', '-w', '-y', and '-b' and have the same semantics as options on the command line except they only apply to the files in the source list. The login name used on the destination host is the same as the local host unless the destination name is of the format ''login@host".

The *notify* command is used to mail the list of files updated (and any errors that may have occured) to the listed names. If no '@' appears in the name, the destination host is appended to the name (e.g., name1@host, name2@host, ...).

The *except* command is used to update all of the files in the source list **except** for the files listed in *name list*. This is usually used to copy everything in a directory except certain files.

The *except_pat* command is like the *except* command except that *pattern list* is a list of regular expressions (see *ed*(1) for details). If one of the patterns matches some string within a file name, that file will be ignored. Note that since '\' is a quote character, it must be doubled to become part of the regular expression. Variables are expanded in *pattern list* but not shell file pattern matching characters. To include a '$', it must be escaped with '\'.

The *special* command is used to specify *sh*(1) commands that are to be executed on the remote host after the file in *name list* is updated or installed. If the *name list* is omitted then the shell commands will be executed for every file updated or installed. The shell variable 'FILE' is set to the current filename before executing the commands in *string*. *String* starts and ends with '"' and can cross multiple lines in *distfile*. Multiple commands to the shell should be separated by ';'. Commands are executed in the user's home directory on the host being updated. The *special* command can be used to rebuild private databases, etc. after a program has been updated.

The following is a small example.

```
HOSTS = ( matisse root@arpa)

FILES = ( /bin /lib /usr/bin /usr/games
        /usr/include/{ *.h,{stand,sys,vax*,pascal,machine}/*.h}
        /usr/lib /usr/man/man? /usr/ucb /usr/local/rdist )

EXLIB = ( Mail.rc aliases aliases.dir aliases.pag crontab dshrc
        sendmail.cf sendmail.fc sendmail.hf sendmail.st uucp vfont )

${FILES} -> ${HOSTS}
        install -R ;
        except /usr/lib/${EXLIB} ;
        except /usr/games/lib ;
        special /usr/lib/sendmail "/usr/lib/sendmail -bz" ;
```

```
srcs:
/usr/src/bin -> arpa
        except_pat ( \\.o\$ /SCCS\$ ) ;

IMAGEN = (ips dviimp catdvi)

imagen:
/usr/local/${IMAGEN} -> arpa
        install /usr/local/lib ;
        notify ralph ;

${FILES} :: stamp.cory
        notify root@cory ;
```

FILES

distfile	input command file
/tmp/rdist*	temporary file for update lists

SEE ALSO

sh(1), csh(1), stat(2)

DIAGNOSTICS

A complaint about mismatch of rdist version numbers may really stem from some problem with starting your shell, e.g., you are in too many groups.

BUGS

Source files must reside on the local host where rdist is executed.

There is no easy way to have a special command executed after all files in a directory have been updated.

Variable expansion only works for name lists; there should be a general macro facility.

Rdist aborts on files which have a negative mtime (before Jan 1, 1970).

There should be a 'force' option to allow replacement of non-empty directories by regular files or symlinks. A means of updating file modes and owners of otherwise identical files is also needed.

NAME

rlogin - remote login

SYNOPSIS

rlogin rhost [**-e** *c*] [**-8**] [**-L**] [**-l** username]
rhost [**-e***c*] [**-8**] [**-L**] [**-l** username]

DESCRIPTION

Rlogin connects your terminal on the current local host system *lhost* to the remote host system *rhost.*

Each host has a file */etc/hosts.equiv* which contains a list of *rhost*'s with which it shares account names. (The host names must be the standard names as described in *remsh*(1C).) When you *rlogin* as the same user on an equivalent host, you don't need to give a password. Each user may also have a private equivalence list in a file .rhosts in his login directory. Each line in this file should contain an *rhost* and a *username* separated by a space, giving additional cases where logins without passwords are to be permitted. If the originating user is not equivalent to the remote user, then a login and password will be prompted for on the remote machine as in *login*(1). To avoid some security problems, the .rhosts file must be owned by either the remote user or root.

The remote terminal type is the same as your local terminal type (as given in your environment TERM variable). The terminal or window size is also copied to the remote system if the server supports the option, and changes in size are reflected as well. All echoing takes place at the remote site, so that (except for delays) the rlogin is transparent. Flow control via ^S and ^Q and flushing of input and output on interrupts are handled properly. The optional argument **-8** allows an eight-bit input data path at all times; otherwise parity bits are stripped except when the remote side's stop and start characters are other than ^S/^Q. The argument **-L** allows the rlogin session to be run in litout mode. A line of the form ``~.'' disconnects from the remote host, where ``~'' is the escape character. Similarly, the line ``~^Z'' (where ^Z, control-Z, is the suspend character) will suspend the rlogin session. Substitution of the delayed-suspend character (normally ^Y) for the suspend character suspends the send portion of the rlogin, but allows output from the remote system. A different escape character may be specified by the **-e** option. There is no space separating this option flag and the argument character.

SEE ALSO

remsh(1C)

FILES

/usr/hosts/*	for *rhost* version of the command

BUGS

More of the environment should be propagated.

NAME

remsh - remote shell

SYNOPSIS

remsh host [**-l** username] [**-n**] command
host [**-l** username] [**-n**] command

DESCRIPTION

Rsh connects to the specified *host,* and executes the specified *command. Rsh* copies its standard input to the remote command, the standard output of the remote command to its standard output, and the standard error of the remote command to its standard error. Interrupt, quit and terminate signals are propagated to the remote command; *remsh* normally terminates when the remote command does.

The remote username used is the same as your local username, unless you specify a different remote name with the -l option. This remote name must be equivalent (in the sense of *rlogin*(1C)) to the originating account; no provision is made for specifying a password with a command.

If you omit *command,* then instead of executing a single command, you will be logged in on the remote host using *rlogin*(1C).

Shell metacharacters which are not quoted are interpreted on local machine, while quoted metacharacters are interpreted on the remote machine. Thus the command

```
remsh otherhost cat remotefile >> localfile
```

appends the remote file *remotefile* to the localfile *localfile,* while

```
remsh otherhost cat remotefile ">>" otherremotefile
```

appends *remotefile* to *otherremotefile.*

Host names are given in the file /etc/hosts. Each host has one standard name (the first name given in the file), which is rather long and unambiguous, and optionally one or more nicknames. The host names for local machines are also commands in the directory /usr/hosts; if you put this directory in your search path then the **remsh** can be omitted.

FILES

/etc/hosts
/usr/hosts/*

SEE ALSO

rlogin(1C)

BUGS

If you are using *csh*(1) and put a *remsh*(1C) in the background without redirecting its input away from the terminal, it will block even if no reads are posted by the remote command. If no input is desired you should redirect the input of *remsh* to /dev/null using the **-n** option.

You cannot run an interactive command (like *rogue*(6) or *vi*(1)); use *rlogin*(1C).

Stop signals stop the local *remsh* process only; this is arguably wrong, but currently hard to fix for reasons too complicated to explain here.

NAME

ruptime - show host status of local machines

SYNOPSIS

ruptime [**-a**] [**-r**] [**-l**] [**-t**] [**-u**]

DESCRIPTION

Ruptime gives a status line like *uptime* for each machine on the local network; these are formed from packets broadcast by each host on the network once a minute.

Machines for which no status report has been received for 11 minutes are shown as being down.

Users idle an hour or more are not counted unless the **-a** flag is given.

Normally, the listing is sorted by host name. The **-l** , **-t** , and **-u** flags specify sorting by load average, uptime, and number of users, respectively. The **-r** flag reverses the sort order.

FILES

/usr/spool/rwho/whod.* data files

SEE ALSO

rwho(1C)

NAME

rwho - who's logged in on local machines

SYNOPSIS

rwho [**-a**]

DESCRIPTION

The *rwho* command produces output similar to *who,* but for all machines on the local network. If no report has been received from a machine for 5 minutes then *rwho* assumes the machine is down, and does not report users last known to be logged into that machine.

If a users hasn't typed to the system for a minute or more, then *rwho* reports this idle time. If a user hasn't typed to the system for an hour or more, then the user will be omitted from the output of *rwho* unless the **-a** flag is given.

FILES

/usr/spool/rwho/whod.* information about other machines

SEE ALSO

ruptime(1C), rwhod(1M)

BUGS

This is unwieldy when the number of machines on the local net is large.

NAME

talk - talk to another user

SYNOPSIS

talk person [ttyname]

DESCRIPTION

Talk is a visual communication program which copies lines from your terminal to that of another user.

If you wish to talk to someone on you own machine, then *person* is just the person's login name. If you wish to talk to a user on another host, then *person* is of the form :

host!user or
host.user or
host:user or
user@host

though *host@user* is perhaps preferred.

If you want to talk to a user who is logged in more than once, the *ttyname* argument may be used to indicate the appropriate terminal name.

When first called, it sends the message

```
Message from TalkDaemon@his_machine...
talk: connection requested by your_name@your_machine.
talk: respond with: talk your_name@your_machine
```

to the user you wish to talk to. At this point, the recipient of the message should reply by typing

```
talk  your_name@your_machine
```

It doesn't matter from which machine the recipient replies, as long as his login-name is the same. Once communication is established, the two parties may type simultaneously, with their output appearing in separate windows. Typing control L will cause the screen to be reprinted, while your erase, kill, and word kill characters will work in talk as normal. To exit, just type your interrupt character; *talk* then moves the cursor to the bottom of the screen and restores the terminal.

Permission to talk may be denied or granted by use of the *mesg* command. At the outset talking is allowed. Certain commands, in particular *nroff* and *pr*(1) disallow messages in order to prevent messy output.

FILES

/etc/hosts	to find the recipient's machine
/etc/utmp	to find the recipient's tty

SEE ALSO

mesg(1), who(1), mail(1), write(1)

NAME

telnet - user interface to the TELNET protocol

SYNOPSIS

telnet [host [port]]

DESCRIPTION

Telnet is used to communicate with another host using the **TELNET** protocol. If *telnet* is invoked without arguments, it enters command mode, indicated by its prompt (''telnet>''). In this mode, it accepts and executes the commands listed below. If it is invoked with arguments, it performs an **open** command (see below) with those arguments.

Once a connection has been opened, *telnet* enters an input mode. The input mode entered will be either ''character at a time'' or ''line by line'' depending on what the remote system supports.

In ''character at a time'' mode, most text typed is immediately sent to the remote host for processing.

In ''line by line'' mode, all text is echoed locally, and (normally) only completed lines are sent to the remote host. The ''local echo character'' (initially ''^E'') may be used to turn off and on the local echo (this would mostly be used to enter passwords without the password being echoed).

In either mode, if the *localchars* toggle is TRUE (the default in line mode; see below), the user's *quit*, *intr*, and *flush* characters are trapped locally, and sent as **TELNET** protocol sequences to the remote side. There are options (see **toggle** *autoflush* and **toggle** *autosynch* below) which cause this action to flush subsequent output to the terminal (until the remote host acknowledges the **TELNET** sequence) and flush previous terminal input (in the case of *quit* and *intr*).

While connected to a remote host, *telnet* command mode may be entered by typing the *telnet* ''escape character'' (initially ''^]''). When in command mode, the normal terminal editing conventions are available.

COMMANDS

The following commands are available. Only enough of each command to uniquely identify it need be typed (this is also true for arguments to the **mode, set, toggle**, and **display** commands).

open *host* [*port*]

Open a connection to the named host. If no port number is specified, *telnet* will attempt to contact a **TELNET** server at the default port. The host specification may be either a host name (see *hosts*(5)) or an Internet address specified in the ''dot notation'' (see *inet*(3N)).

close

Close a **TELNET** session and return to command mode.

quit

Close any open **TELNET** session and exit *telnet*. An end of file (in command mode) will also close a session and exit.

z

Suspend *telnet*. This command only works when the user is using the *csh*(1).

mode *type*
: *Type* is either *line* (for "line by line" mode) or *character* (for "character at a time" mode). The remote host is asked for permission to go into the requested mode. If the remote host is capable of entering that mode, the requested mode will be entered.

status
: Show the current status of *telnet*. This includes the peer one is connected to, as well as the current mode.

display [*argument...*]
: Displays all, or some, of the **set** and **toggle** values (see below).

? [*command*]
: Get help. With no arguments, *telnet* prints a help summary. If a command is specified, *telnet* will print the help information for just that command.

send *arguments*
: Sends one or more special character sequences to the remote host. The following are the arguments which may be specified (more than one argument may be specified at a time):

escape
: Sends the current *telnet* escape character (initially "^]").

synch
: Sends the **TELNET SYNCH** sequence. This sequence causes the remote system to discard all previously typed (but not yet read) input. This sequence is sent as TCP urgent data (and may not work if the remote system is a 4.2 BSD system -- if it doesn't work, a lower case "r" may be echoed on the terminal).

brk
: Sends the **TELNET BRK** (Break) sequence, which may have significance to the remote system.

ip
: Sends the **TELNET IP** (Interrupt Process) sequence, which should cause the remote system to abort the currently running process.

ao
: Sends the **TELNET AO** (Abort Output) sequence, which should cause the remote system to flush all output **from** the remote system **to** the user's terminal.

ayt
: Sends the **TELNET AYT** (Are You There) sequence, to which the remote system may or may not choose to respond.

ec
: Sends the **TELNET EC** (Erase Character) sequence, which should cause the remote system to erase the last character entered.

el
: Sends the **TELNET EL** (Erase Line) sequence, which should cause the remote system to erase the line currently being entered.

ga
: Sends the **TELNET GA** (Go Ahead) sequence, which likely has no

significance to the remote system.

nop

Sends the **TELNET NOP** (No OPeration) sequence.

?

Prints out help information for the **send** command.

set *argument value*

Set any one of a number of *telnet* variables to a specific value. The special value "off" turns off the function associated with the variable. The values of variables may be interrogated with the **display** command. The variables which may be specified are:

echo

This is the value (initially "^E") which, when in "line by line" mode, toggles between doing local echoing of entered characters (for normal processing), and suppressing echoing of entered characters (for entering, say, a password).

escape

This is the *telnet* escape character (initially "^[") which causes entry into *telnet* command mode (when connected to a remote system).

interrupt

If *telnet* is in *localchars* mode (see **toggle** *localchars* below) and the *interrupt* character is typed, a **TELNET IP** sequence (see **send** *ip* above) is sent to the remote host. The initial value for the interrupt character is taken to be the terminal's **intr** character.

quit

If *telnet* is in *localchars* mode (see **toggle** *localchars* below) and the *quit* character is typed, a **TELNET BRK** sequence (see **send** *brk* above) is sent to the remote host. The initial value for the quit character is taken to be the terminal's **quit** character.

flushoutput

If *telnet* is in *localchars* mode (see **toggle** *localchars* below) and the *flushoutput* character is typed, a **TELNET AO** sequence (see **send** *ao* above) is sent to the remote host. The initial value for the flush character is taken to be the terminal's **flush** character.

erase

If *telnet* is in *localchars* mode (see **toggle** *localchars* below), **and** if *telnet* is operating in "character at a time" mode, then when this character is typed, a **TELNET EC** sequence (see **send** *ec* above) is sent to the remote system. The initial value for the erase character is taken to be the terminal's **erase** character.

kill

If *telnet* is in *localchars* mode (see **toggle** *localchars* below), **and** if *telnet* is operating in "character at a time" mode, then when this character is typed, a **TELNET EL** sequence (see **send** *el* above) is sent to the remote system. The initial value for the kill character is taken to be the terminal's **kill** character.

eof

If *telnet* is operating in "line by line" mode, entering this character as the first character on a line will cause this character to be sent to the remote system. The initial value of the eof character is taken to be the terminal's **eof** character.

toggle *arguments...*

Toggle (between TRUE and FALSE) various flags that control how *telnet* responds to events. More than one argument may be specified. The state of these flags may be interrogated with the **display** command. Valid arguments are:

localchars

If this is TRUE, then the *flush*, *interrupt*, *quit*, *erase*, and *kill* characters (see **set** above) are recognized locally, and transformed into (hopefully) appropriate **TELNET** control sequences (respectively *ao*, *ip*, *brk*, *ec*, and *el*; see **send** above). The initial value for this toggle is TRUE in ''line by line'' mode, and FALSE in ''character at a time'' mode.

autoflush

If *autoflush* and *localchars* are both TRUE, then when the *ao*, *intr*, or *quit* characters are recognized (and transformed into **TELNET** sequences; see **set** above for details), *telnet* refuses to display any data on the user's terminal until the remote system acknowledges (via a **TELNET** *Timing Mark* option) that it has processed those **TELNET** sequences. The initial value for this toggle is TRUE if the terminal user had not done an "stty noflsh", otherwise FALSE (see *stty(1)).*

autosynch

If *autosynch* and *localchars* are both TRUE, then when either the *intr* or *quit* characters is typed (see **set** above for descriptions of the *intr* and *quit* characters), the resulting **TELNET** sequence sent is followed by the **TELNET SYNCH** sequence. This procedure **should** cause the remote system to begin throwing away all previously typed input until both of the **TELNET** sequences have been read and acted upon. The initial value of this toggle is FALSE.

crmod

Toggle carriage return mode. When this mode is enabled, most carriage return characters received from the remote host will be mapped into a carriage return followed by a line feed. This mode does not affect those characters typed by the user, only those received from the remote host. This mode is not very useful unless the remote host only sends carriage return, but never line feed. The initial value for this toggle is FALSE.

debug

Toggles socket level debugging (useful only to the *super*user*)*. The initial value for this toggle is FALSE.

options

Toggles the display of some internal *telnet* protocol processing (having to do with **TELNET** options). The initial value for this toggle is FALSE.

netdata

Toggles the display of all network data (in hexadecimal format). The initial value for this toggle is FALSE.

?

Displays the legal **toggle** commands.

BUGS

There is no adequate way for dealing with flow control.

On some remote systems, echo has to be turned off manually when in ''line by line'' mode.

There is enough settable state to justify a *.telnetrc* file.

No capability for a *.telnetrc* file is provided.

In ''line by line'' mode, the terminal's *eof* character is only recognized (and sent to the remote system) when it is the first character on a line.

NAME

tftp - trivial file transfer program

SYNOPSIS

tftp [host]

DESCRIPTION

Tftp is the user interface to the Internet TFTP (Trivial File Transfer Protocol), which allows users to transfer files to and from a remote machine. The remote *host* may be specified on the command line, in which case *tftp* uses *host* as the default host for future transfers (see the **connect** command below).

COMMANDS

Once *tftp* is running, it issues the prompt **tftp>** and recognizes the following commands:

connect *host-name* [*port*]
: Set the *host* (and optionally *port*) for transfers. Note that the TFTP protocol, unlike the FTP protocol, does not maintain connections betweeen transfers; thus, the *connect* command does not actually create a connection, but merely remembers what host is to be used for transfers. You do not have to use the *connect* command; the remote host can be specified as part of the *get* or *put* commands.

mode *transfer-mode*
: Set the mode for transfers; *transfer-mode* may be one of *ascii* or *binary*. The default is *ascii*.

put *file*

put *localfile remotefile*
put *file1 file2 ... fileN remote-directory*
: Put a file or set of files to the specified remote file or directory. The destination can be in one of two forms: a filename on the remote host, if the host has already been specified, or a string of the form *host:filename* to specify both a host and filename at the same time. If the latter form is used, the hostname specified becomes the default for future transfers. If the remote-directory form is used, the remote host is assumed to be a *UNIX* machine.

get *filename*
get *remotename localname*
get *file1 file2 ... fileN*
: Get a file or set of files from the specified *sources*. *Source* can be in one of two forms: a filename on the remote host, if the host has already been specified, or a string of the form *host:filename* to specify both a host and filename at the same time. If the latter form is used, the last hostname specified becomes the default for future transfers.

quit Exit *tftp*. An end of file also exits.

verbose
: Toggle verbose mode.

trace Toggle packet tracing.

status Show current status.

rexmt *retransmission-timeout*
Set the per-packet retransmission timeout, in seconds.

timeout *total-transmission-timeout*
Set the total transmission timeout, in seconds.

ascii Shorthand for "mode ascii"

binary Shorthand for "mode binary"

? [*command-name* ...]
Print help information.

BUGS

Because there is no user-login or validation within the *TFTP* protocol, the remote site will probably have some sort of file-access restrictions in place. The exact methods are specific to each site and therefore difficult to document here.

NAME

ftpd - DARPA Internet File Transfer Protocol server

SYNOPSIS

/etc/ftpd [**-d**] [**-l**] [**-t**timeout]

DESCRIPTION

Ftpd is the DARPA Internet File Transfer Protocol server process. The server uses the TCP protocol and listens at the port specified in the ''ftp'' service specification; see *services* (5).

If the **-d** option is specified, debugging information is written to the syslog.

If the **-l** option is specified, each ftp session is logged in the syslog.

The ftp server will timeout an inactive session after 15 minutes. If the **-t** option is specified, the inactivity timeout period will be set to *timeout*.

The ftp server currently supports the following ftp requests; case is not distinguished.

Request	**Description**
ABOR	abort previous command
ACCT	specify account (ignored)
ALLO	allocate storage (vacuously)
APPE	append to a file
CDUP	change to parent of current working directory
CWD	change working directory
DELE	delete a file
HELP	give help information
LIST	give list files in a directory (''ls -lg'')
MKD	make a directory
MODE	specify data transfer *mode*
NLST	give name list of files in directory (''ls'')
NOOP	do nothing
PASS	specify password
PASV	prepare for server-to-server transfer
PORT	specify data connection port
PWD	print the current working directory
QUIT	terminate session
RETR	retrieve a file
RMD	remove a directory
RNFR	specify rename-from file name
RNTO	specify rename-to file name
STOR	store a file
STOU	store a file with a unique name
STRU	specify data transfer *structure*
TYPE	specify data transfer *type*
USER	specify user name
XCUP	change to parent of current working directory
XCWD	change working directory
XMKD	make a directory
XPWD	print the current working directory
XRMD	remove a directory

The remaining ftp requests specified in Internet RFC 959 are recognized, but not implemented.

The ftp server will abort an active file transfer only when the ABOR command is preceded by a Telnet "Interrupt Process" (IP) signal and a Telnet "Synch" signal in the command Telnet stream, as described in Internet RFC 959.

Ftpd interprets file names according to the ``globbing'' conventions used by *csh*(1). This allows users to utilize the metacharacters ``*?[]{}~''.

Ftpd authenticates users according to three rules.

1) The user name must be in the password data base, */etc/passwd*, and not have a null password. In this case a password must be provided by the client before any file operations may be performed.

2) The user name must not appear in the file */etc/ftpusers*.

3) The user must have a standard shell returned by *getusershell*(3).

4) If the user name is ``anonymous'' or ``ftp'', an anonymous ftp account must be present in the password file (user ``ftp''). In this case the user is allowed to log in by specifying any password (by convention this is given as the client host's name).

In the last case, *ftpd* takes special measures to restrict the client's access privileges. The server performs a *chroot*(2) command to the home directory of the ``ftp'' user. In order that system security is not breached, it is recommended that the ``ftp'' subtree be constructed with care; the following rules are recommended.

~ftp) Make the home directory owned by ``ftp'' and unwritable by anyone.

~ftp/bin)
Make this directory owned by the super-user and unwritable by anyone. The program *ls*(1) must be present to support the list commands. This program should have mode 111.

~ftp/etc)
Make this directory owned by the super-user and unwritable by anyone. The files *passwd*(4) and *group*(4) must be present for the *ls* command to work properly. These files should be mode 444.

~ftp/pub)
Make this directory mode 777 and owned by ``ftp''. Users should then place files which are to be accessible via the anonymous account in this directory.

SEE ALSO

ftp(1C), getusershell(3), syslogd(1M)

BUGS

The anonymous account is inherently dangerous and should avoided when possible.

The server must run as the super-user to create sockets with privileged port numbers. It maintains an effective user id of the logged in user, reverting to the super-user only when binding addresses to sockets. The possible security holes have been extensively scrutinized, but are possibly incomplete.

NAME

gettable - get NIC format host tables from a host

SYNOPSIS

/etc/gettable [**-v**] *host* [outfile]

DESCRIPTION

Gettable is a simple program used to obtain the NIC standard host tables from a ``nicname'' server. The indicated *host* is queried for the tables. The tables, if retrieved, are placed in the file *outfile* or by default, *hosts.txt*.

The **-v** option will get just the version number instead of the complete host table and put the output in the file *outfile* or by default, *hosts.ver*.

Gettable operates by opening a TCP connection to the port indicated in the service specification for ``nicname''. A request is then made for ``ALL'' names and the resultant information is placed in the output file.

Gettable is best used in conjunction with the *htable*(1M) program which converts the NIC standard file format to that used by the network library lookup routines.

NOTE: When connecting directly to the ARPA internet network, refer to the information on page 4-5, ``Regenerating /etc/hosts and /etc/networks'', in the ICON/UXV-NET Networking Tools Guide.

SEE ALSO

intro(3N), htable(1M), named(1M)

BUGS

If the name-domain system provided network name mapping well as host name mapping, *gettable* would no longer be needed.

NAME

htable - convert NIC standard format host tables

SYNOPSIS

/etc/htable [**-c** *connected-nets*] [**-l** *local-nets*] *file*

DESCRIPTION

Htable is used to convert host files in the format specified in Internet RFC 810 to the format used by the network library routines. Three files are created as a result of running *htable*: *hosts*, *networks*, and *gateways*. The *hosts* file may be used by the *gethostbyname*(3N) routines in mapping host names to addresses if the nameserver, *named*(1M), is not used. The *networks* file is used by the *getnetent*(3N) routines in mapping network names to numbers. The *gateways* file may be used by the routing daemon in identifying ''passive'' Internet gateways; see *routed*(1M) for an explanation.

If any of the files *localhosts*, *localnetworks*, or *localgateways* are present in the current directory, the file's contents is prepended to the output file. Of these, only the gateways file is interpreted. This allows sites to maintain local aliases and entries which are not normally present in the master database. Only one gateway to each network will be placed in the gateways file; a gateway listed in the localgateways file will override any in the input file.

If the gateways file is to be used, a list of networks to which the host is directly connected is specified with the **-c** flag. The networks, separated by commas, may be given by name or in Internet-standard dot notation, e.g. **-c** arpanet,128.32,local-ether-net. *Htable* only includes gateways which are directly connected to one of the networks specified, or which can be reached from another gateway on a connected net.

If the **-l** option is given with a list of networks (in the same format as for **-c**), these networks will be treated as ''local,'' and information about hosts on local networks is taken only from the localhosts file. Entries for local hosts from the main database will be omitted. This allows the localhosts file to completely override any entries in the input file.

Htable is best used in conjunction with the *gettable*(1M) program which retrieves the NIC database from a host.

SEE ALSO

intro(3N), gettable(1M), named(1M)

BUGS

If the name-domain system provided network name mapping well as host name mapping, *htable* would no longer be needed.

NAME

ifconfig - configure network interface parameters

SYOPNSIS

/etc/ifconfig interface address_family [*address* [*dest_address*]] [*parameters*]
/etc/ifconfig interface [protocol_family]

DESCRIPTION

Ifconfig is used to assign an address to a network interface and/or configure network interface parameters. *Ifconfig* must be used at boot time to define the network address of each interface present on a machine; it may also be used at a later time to redefine an interface's address or other operating parameters. The *interface* parameter is a string of the form ''name unit'', e.g. ''ex0''.

Since an interface may receive transmissions in differing protocols, each of which may require separate naming schemes, it is necessary to specify the *address_family*, which may change the interpretation of the remaining parameters. The address families currently supported are ''inet'' and ''ns''.

For the DARPA-Internet family, the address is either a host name present in the host name data base, *hosts*(4), or a DARPA Internet address expressed in the Internet standard ''dot notation''. For the Xerox Network Systems(tm) family, addresses are *net:a.b.c.d.e.f*, where *net* is the assigned network number (in decimal), and each of the six bytes of the host number, *a* through *f*, are specified in hexadecimal. The host number may be omitted on 10Mb/s Ethernet interfaces, which use the hardware physical address, and on interfaces other than the first.

The following parameters may be set with *ifconfig*:

up Mark an interface ''up''. This may be used to enable an interface after an ''ifconfig down.'' It happens automatically when setting the first address on an interface. If the interface was reset when previously marked down, the hardware will be re-initialized.

down Mark an interface ''down''. When an interface is marked ''down'', the system will not attempt to transmit messages through that interface. If possible, the interface will be reset to disable reception as well. This action does not automatically disable routes using the interface.

trailers Request the use of a ''trailer'' link level encapsulation when sending (default). If a network interface supports *trailers*, the system will, when possible, encapsulate outgoing messages in a manner which minimizes the number of memory to memory copy operations performed by the receiver. On networks that support the Address Resolution Protocol (see *arp*(7P); currently, only 10 Mb/s Ethernet), this flag indicates that the system should request that other systems use trailers when sending to this host. Similarly, trailer encapsulations will be sent to other hosts that have made such requests. Currently used by Internet protocols only.

-trailers Disable the use of a ''trailer'' link level encapsulation.

arp Enable the use of the Address Resolution Protocol in mapping between network level addresses and link level addresses (default). This is currently implemented for mapping between DARPA Internet addresses and 10Mb/s Ethernet addresses.

-arp	Disable the use of the Address Resolution Protocol.
metric *n*	Set the routing metric of the interface to *n*, default 0. The routing metric is used by the routing protocol (*routed*(1M)). Higher metrics have the effect of making a route less favorable; metrics are counted as addition hops to the destination network or host.
debug	Enable driver dependent debugging code; usually, this turns on extra console error logging.
-debug	Disable driver dependent debugging code.
netmask *mask*	(Inet only) Specify how much of the address to reserve for subdividing networks into sub-networks. The mask includes the network part of the local address and the subnet part, which is taken from the host field of the address. The mask can be specified as a single hexadecimal number with a leading 0x, with a dot-notation Internet address, or with a pseudo-network name listed in the network table *networks*(4). The mask contains 1's for the bit positions in the 32-bit address which are to be used for the network and subnet parts, and 0's for the host part. The mask should contain at least the standard network portion, and the subnet field should be contiguous with the network portion.
dstaddr	Specify the address of the correspondent on the other end of a point to point link.
broadcast	(Inet only) Specify the address to use to represent broadcasts to the network. The default broadcast address is the address with a host part of all 1's.

Ifconfig displays the current configuration for a network interface when no optional parameters are supplied. If a protocol family is specified, Ifconfig will report only the details specific to that protocol family.

Only the super-user may modify the configuration of a network interface.

DIAGNOSTICS

Messages indicating the specified interface does not exit, the requested address is unknown, or the user is not privileged and tried to alter an interface's configuration.

SEE ALSO

netstat(1), intro(7N), rc(1M)

NAME

inetd - internet ''super-server''

SYNOPSIS

/etc/inetd [**-d**] [configuration file]

DESCRIPTION

Inetd should be run at boot time by */etc/rc.local*. It then listens for connections on certain internet sockets. When a connection is found on one of its sockets, it decides what service the socket corresponds to, and invokes a program to service the request. After the program is finished, it continues to listen on the socket (except in some cases which will be described below). Essentially, *inetd* allows running one daemon to invoke several others, reducing load on the system.

Upon execution, *inetd* reads its configuration information from a configuration file which, by default, is */etc/inetd.conf*. There must be an entry for each field of the configuration file, with entries for each field separated by a tab or a space. Comments are denoted by a ''#'' at the beginning of a line. There must be an entry for each field. The fields of the configuration file are as follows:

service name
socket type
protocol
wait/nowait
user
server program
server program arguments

The *service name* entry is the name of a valid service in the file */etc/services/*. For ''internal'' services (discussed below), the service name *must* be the official name of the service (that is, the first entry in */etc/services*).

The *socket type* should be one of ''stream'', ''dgram'', ''raw'', ''rdm'', or ''seqpacket'', depending on whether the socket is a stream, datagram, raw, reliably delivered message, or sequenced packet socket.

The *protocol* must be a valid protocol as given in */etc/protocols*. Examples might be ''tcp'' or ''udp''.

The *wait/nowait* entry is applicable to datagram sockets only (other sockets should have a ''nowait'' entry in this space). If a datagram server connects to its peer, freeing the socket so *inetd* can received further messages on the socket, it is said to be a ''multi-threaded'' server, and should use the ''nowait'' entry. For datagram servers which process all incoming datagrams on a socket and eventually time out, the server is said to be ''single-threaded'' and should use a ''wait'' entry. ''Comsat'' (''biff'') and ''talk'' are both examples of the latter type of datagram server. *Tftpd* is an exception; it is a datagram server that establishes pseudo-connections. It must be listed as ''wait'' in order to avoid a race; the server reads the first packet, creates a new socket, and then forks and exits to allow *inetd* to check for new service requests to spawn new servers.

The *user* entry should contain the user name of the user as whom the server should run. This allows for servers to be given less permission than root. The *server program* entry should contain the pathname of the program which is to be executed by *inetd* when a request is found on its socket. If *inetd* provides this service internally, this entry should be ''internal''.

The arguments to the server program should be just as they normally are, starting with argv[0], which is the name of the program. If the service is provided internally, the word ''internal'' should take the place of this entry.

Inetd provides several ''trivial'' services internally by use of routines within itself. These services are ''echo'', ''discard'', ''chargen'' (character generator), ''daytime'' (human readable time), and ''time'' (machine readable time, in the form of the number of seconds since midnight, January 1, 1900). All of these services are tcp based. For details of these services, consult the appropriate RFC from the Network Information Center.

Inetd rereads its configuration file when it receives a hangup signal, SIGHUP. Services may be added, deleted or modified when the configuration file is reread.

SEE ALSO

comsat(1M), ftpd(1M), rexecd(1M), rlogind(1M), remshd(1M), telnetd(1M), tftpd(1M)

NAME

ping - send ICMP ECHO_REQUEST packets to network hosts

SYNOPSIS

/etc/ping [**-r**] [**-v**] *host* [*packetsize*] [*count*]

DESCRIPTION

The DARPA Internet is a large and complex aggregation of network hardware, connected together by gateways. Tracking a single-point hardware or software failure can often be difficult. *Ping* utilizes the ICMP protocol's mandatory ECHO_REQUEST datagram to elicit an ICMP ECHO_RESPONSE from a host or gateway. ECHO_REQUEST datagrams (''pings'') have an IP and ICMP header, followed by a **struct timeval**, and then an arbitrary number of ''pad'' bytes used to fill out the packet. Default datagram length is 64 bytes, but this may be changed using the command-line option. Other options are:

-r Bypass the normal routing tables and send directly to a host on an attached network. If the host is not on a directly-attached network, an error is returned. This option can be used to ping a local host through an interface that has no route through it (e.g., after the interface was dropped by *routed*(1M)).

-v Verbose output. ICMP packets other than ECHO RESPONSE that are received are listed.

When using *ping* for fault isolation, it should first be run on the local host, to verify that the local network interface is up and running. Then, hosts and gateways further and further away should be ''pinged''. *Ping* sends one datagram per second, and prints one line of output for every ECHO_RESPONSE returned. No output is produced if there is no response. If an optional *count* is given, only that number of requests is sent. Round-trip times and packet loss statistics are computed. When all responses have been received or the program times out (with a *count* specified), or if the program is terminated with a SIGINT, a brief summary is displayed.

This program is intended for use in network testing, measurement and management. It should be used primarily for manual fault isolation. Because of the load it could impose on the network, it is unwise to use *ping* during normal operations or from automated scripts.

AUTHOR

Mike Muuss

SEE ALSO

netstat(1), ifconfig(1M)

NAME

remshd - remote shell server

SYNOPSIS

/etc/remshd

DESCRIPTION

Rshd is the server for the *rcmd*(3) routine and, consequently, for the *remsh*(1C) program. The server provides remote execution facilities with authentication based on privileged port numbers from trusted hosts.

Rshd listens for service requests at the port indicated in the ''cmd'' service specification; see *services*(4). When a service request is received the following protocol is initiated:

1) The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.

2) The server reads characters from the socket up to a null ('\0') byte. The resultant string is interpreted as an ASCII number, base 10.

3) If the number received in step 1 is non-zero, it is interpreted as the port number of a secondary stream to be used for the **stderr**. A second connection is then created to the specified port on the client's machine. The source port of this second connection is also in the range 0-1023.

4) The server checks the client's source address and requests the corresponding host name (see *gethostbyaddr*(3N), *hosts*(4) and *named*(1M)). If the hostname cannot be determined, the dot-notation representation of the host address is used.

5) A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as the user identity on the **client**'s machine.

6) A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as a user identity to use on the **server**'s machine.

7) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.

8) *Rshd* then validates the user according to the following steps. The local (server-end) user name is looked up in the password file and a *chdir* is performed to the user's home directory. If either the lookup or *chdir* fail, the connection is terminated. If the user is not the super-user, (user id 0), the file */etc/hosts.equiv* is consulted for a list of hosts considered ''equivalent''. If the client's host name is present in this file, the authentication is considered successful. If the lookup fails, or the user is the super-user, then the file *.rhosts* in the home directory of the remote user is checked for the machine name and identity of the user on the client's machine. If this lookup fails, the connection is terminated.

9) A null byte is returned on the initial socket and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *remshd*.

DIAGNOSTICS

Except for the last one listed below, all diagnostic messages are returned on the initial socket, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 9 above upon successful completion of all the steps prior to the execution of the login shell).

"locuser too long"
The name of the user on the client's machine is longer than 16 characters.

"remuser too long"
The name of the user on the remote machine is longer than 16 characters.

"command too long "
The command line passed exceeds the size of the argument list (as configured into the system).

"Login incorrect."
No password file entry for the user name existed.

"No remote directory."
The *chdir* command to the home directory failed.

"Permission denied."
The authentication procedure described above failed.

"Can't make pipe."
The pipe needed for the **stderr**, wasn't created.

"Try again."
A *fork* by the server failed.

"<shellname>: ..."
The user's login shell could not be started. This message is returned on the connection associated with the **stderr**, and is not preceded by a flag byte.

SEE ALSO

remsh(1C), rcmd(3)

BUGS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

A facility to allow all data exchanges to be encrypted should be present.

A more extensible protocol should be used.

NAME

rexecd - remote execution server

SYNOPSIS

/etc/rexecd

DESCRIPTION

Rexecd is the server for the *rexec*(3) routine. The server provides remote execution facilities with authentication based on user names and passwords.

Rexecd listens for service requests at the port indicated in the "exec" service specification; see *services*(4). When a service request is received the following protocol is initiated:

1) The server reads characters from the socket up to a null ('\0') byte. The resultant string is interpreted as an ASCII number, base 10.

2) If the number received in step 1 is non-zero, it is interpreted as the port number of a secondary stream to be used for the **stderr**. A second connection is then created to the specified port on the client's machine.

3) A null terminated user name of at most 16 characters is retrieved on the initial socket.

4) A null terminated, unencrypted password of at most 16 characters is retrieved on the initial socket.

5) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.

6) *Rexecd* then validates the user as is done at login time and, if the authentication was successful, changes to the user's home directory, and establishes the user and group protections of the user. If any of these steps fail the connection is aborted with a diagnostic message returned.

7) A null byte is returned on the initial socket and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *rexecd*.

DIAGNOSTICS

Except for the last one listed below, all diagnostic messages are returned on the initial socket, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 7 above upon successful completion of all the steps prior to the command execution).

"username too long"
The name is longer than 16 characters.

"password too long"
The password is longer than 16 characters.

"command too long "
The command line passed exceeds the size of the argument list (as configured into the system).

"Login incorrect."
No password file entry for the user name existed.

"Password incorrect."
The wrong was password supplied.

"No remote directory."
The *chdir* command to the home directory failed.

"Try again."
A *fork* by the server failed.

"<shellname>: ..."
The user's login shell could not be started. This message is returned on the connection associated with the **stderr**, and is not preceded by a flag byte.

SEE ALSO

rexec(3)

BUGS

Indicating "Login incorrect" as opposed to "Password incorrect" is a security breach which allows people to probe a system for users with null passwords.

A facility to allow all data and password exchanges to be encrypted should be present.

NAME

rlogind - remote login server

SYNOPSIS

/etc/rlogind [**-d**]

DESCRIPTION

Rlogind is the server for the *rlogin*(1C) program. The server provides a remote login facility with authentication based on privileged port numbers from trusted hosts.

Rlogind listens for service requests at the port indicated in the ''login'' service specification; see *services*(4). When a service request is received the following protocol is initiated:

1) The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.

2) The server checks the client's source address and requests the corresponding host name (see *gethostbyaddr*(3N), *hosts*(4) and *named*(1M)). If the hostname cannot be determined, the dot-notation representation of the host address is used.

Once the source port and address have been checked, *rlogind* allocates a pseudo terminal (see *pty*(4)), and manipulates file descriptors so that the slave half of the pseudo terminal becomes the **stdin** , **stdout** , and **stderr** for a login process. The login process is an instance of the *login*(1) program, invoked with the **-r** option. The login process then proceeds with the authentication process as described in *remshd*(1M), but if automatic authentication fails, it reprompts the user to login as one finds on a standard terminal line.

The parent of the login process manipulates the master side of the pseduo terminal, operating as an intermediary between the login process and the client instance of the *rlogin* program. In normal operation, the packet protocol described in *pty*(4) is invoked to provide ^S/^Q type facilities and propagate interrupt signals to the remote programs. The login process propagates the client terminal's baud rate and terminal type, as found in the environment variable, ''TERM''; see *environ*(7). The screen or window size of the terminal is requested from the client, and window size changes from the client are propagated to the pseudo terminal.

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1.

''Try again.''
A *fork* by the server failed.

''/bin/sh: ...''
The user's login shell could not be started.

BUGS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an ''open'' environment.

A facility to allow all data exchanges to be encrypted should be present.

A more extensible protocol should be used.

NAME

route - manually manipulate the routing tables

SYNOPSIS

/etc/route [**-f**] [**-n**] [*command args*]

DESCRIPTION

Route is a program used to manually manipulate the network routing tables. It normally is not needed, as the system routing table management daemon, *routed*(1M), should tend to this task.

Route accepts two commands: *add*, to add a route, and *delete*, to delete a route.

All commands have the following syntax:

/etc/route *command* [**net** | **host**] *destination gateway* [*metric*]

where *destination* is the destination host or network, *gateway* is the next-hop gateway to which packets should be addressed, and *metric* is a count indicating the number of hops to the *destination*. The metric is required for *add* commands; it must be zero if the destination is on a directly-attached network, and nonzero if the route utilizes one or more gateways. If adding a route with metric 0, the gateway given is the address of this host on the common network, indicating the interface to be used for transmission. Routes to a particular host are distinguished from those to a network by interpreting the Internet address associated with *destination*. The optional keywords **net** and **host** force the destination to be interpreted as a network or a host, respectively. Otherwise, if the *destination* has a ''local address part'' of INADDR_ANY, or if the *destination* is the symbolic name of a network, then the route is assumed to be to a network; otherwise, it is presumed to be a route to a host. If the route is to a destination connected via a gateway, the *metric* should be greater than 0. All symbolic names specified for a *destination* or *gateway* are looked up first as a host name using *gethostbyname*(3N). If this lookup fails, *getnetbyname*(3N) is then used to interpret the name as that of a network.

Route uses a raw socket and the SIOCADDRT and SIOCDELRT *ioctl*'s to do its work. As such, only the super-user may modify the routing tables.

If the **-f** option is specified, *route* will ''flush'' the routing tables of all gateway entries. If this is used in conjunction with one of the commands described above, the tables are flushed prior to the command's application.

The **-n** option prevents attempts to print host and network names symbolically when reporting actions.

DIAGNOSTICS

''**add [host | network] %s: gateway %s flags %x**''
The specified route is being added to the tables. The values printed are from the routing table entry supplied in the *ioctl* call. If the gateway address used was not the primary address of the gateway (the first one returned by *gethostbyname*), the gateway address is printed numerically as well as symbolically.

''**delete [host | network] %s: gateway %s flags %x**''
As above, but when deleting an entry.

"%s %s done"
When the **-f** flag is specified, each routing table entry deleted is indicated with a message of this form.

"Network is unreachable"
An attempt to add a route failed because the gateway listed was not on a directly-connected network. The next-hop gateway must be given.

"not in table"
A delete operation was attempted for an entry which wasn't present in the tables.

"routing table overflow"
An add operation was attempted, but the system was low on resources and was unable to allocate memory to create the new entry.

SEE ALSO

intro(7N), routed(1M),

NAME

routed - network routing daemon

SYNOPSIS

/etc/routed [**-d**] [**-g**] [**-s**] [**-q**] [**-t**] [*logfile*]

DESCRIPTION

Routed is invoked at boot time to manage the network routing tables. The routing daemon uses a variant of the Xerox NS Routing Information Protocol in maintaining up to date kernel routing table entries. It used a generalized protocol capable of use with multiple address types, but is currently used only for Internet routing within a cluster of networks.

In normal operation *routed* listens on the *udp*(7P) socket for the *route* service (see *services*(4)) for routing information packets. If the host is an internetwork router, it periodically supplies copies of its routing tables to any directly connected hosts and networks.

When *routed* is started, it uses the SIOCGIFCONF *ioctl* to find those directly connected interfaces configured into the system and marked ''up'' (the software loopback interface is ignored). If multiple interfaces are present, it is assumed that the host will forward packets between networks. *Routed* then transmits a *request* packet on each interface (using a broadcast packet if the interface supports it) and enters a loop, listening for *request* and *response* packets from other hosts.

When a *request* packet is received, *routed* formulates a reply based on the information maintained in its internal tables. The *response* packet generated contains a list of known routes, each marked with a ''hop count'' metric (a count of 16, or greater, is considered ''infinite''). The metric associated with each route returned provides a metric *relative to the sender*.

Response packets received by *routed* are used to update the routing tables if one of the following conditions is satisfied:

(1) No routing table entry exists for the destination network or host, and the metric indicates the destination is ''reachable'' (i.e. the hop count is not infinite).

(2) The source host of the packet is the same as the router in the existing routing table entry. That is, updated information is being received from the very internetwork router through which packets for the destination are being routed.

(3) The existing entry in the routing table has not been updated for some time (defined to be 90 seconds) and the route is at least as cost effective as the current route.

(4) The new route describes a shorter route to the destination than the one currently stored in the routing tables; the metric of the new route is compared against the one stored in the table to decide this.

When an update is applied, *routed* records the change in its internal tables and updates the kernel routing table. The change is reflected in the next *response* packet sent.

In addition to processing incoming packets, *routed* also periodically checks the routing table entries. If an entry has not been updated for 3 minutes, the entry's metric is set to infinity and marked for deletion. Deletions are delayed an additional 60 seconds to insure the invalidation is propagated throughout the local internet.

Hosts acting as internetwork routers gratuitously supply their routing tables every 30 seconds to all directly connected hosts and networks. The response is sent to the broadcast address on nets capable of that function, to the destination address on point-to-point links, and to the router's own address on other networks. The normal routing tables are bypassed when sending gratuitous responses. The reception of responses on each network is used to determine that the network and interface are functioning correctly. If no response is received on an interface, another route may be chosen to route around the interface, or the route may be dropped if no alternative is available.

Routed supports several options:

-d	Enable additional debugging information to be logged, such as bad packets received.
-g	This flag is used on internetwork routers to offer a route to the ''default'' destination. This is typically used on a gateway to the Internet, or on a gateway that uses another routing protocol whose routes are not reported to other local routers.
-s	Supplying this option forces *routed* to supply routing information whether it is acting as an internetwork router or not. This is the default if multiple network interfaces are present, or if a point-to-point link is in use.
-q	This is the opposite of the -s option.
-t	If the -t option is specified, all packets sent or received are printed on the standard output. In addition, *routed* will not divorce itself from the controlling terminal so that interrupts from the keyboard will kill the process.

Any other argument supplied is interpreted as the name of file in which *routed*'s actions should be logged. This log contains information about any changes to the routing tables and, if not tracing all packets, a history of recent messages sent and received which are related to the changed route.

In addition to the facilities described above, *routed* supports the notion of ''distant'' *passive* and *active* gateways. When *routed* is started up, it reads the file */etc/gateways* to find gateways which may not be located using only information from the SIOGIFCONF *ioctl*. Gateways specified in this manner should be marked passive if they are not expected to exchange routing information, while gateways marked active should be willing to exchange routing information (i.e. they should have a *routed* process running on the machine). Passive gateways are maintained in the routing tables forever and information regarding their existence is included in any routing information transmitted. Active gateways are treated equally to network interfaces. Routing information is distributed to the gateway and if no routing information is received for a period of the time, the associated route is deleted. External gateways are also passive, but are not placed in the kernel routing table nor are they included in routing updates. The function of external entries is to inform *routed* that another routing process will install such a route, and that alternate routes to that destination should not be installed. Such entries are only required when both routers may learn of routes to the same destination.

The */etc/gateways* is comprised of a series of lines, each in the following format:

< **net** | **host** > *name1* **gateway** *name2* **metric** *value* < **passive** | **active** | **external** >

The **net** or **host** keyword indicates if the route is to a network or specific host.

Name1 is the name of the destination network or host. This may be a symbolic name located in */etc/networks* or */etc/hosts* (or, if started after *named*(1M), known to the name server), or an Internet address specified in ''dot'' notation; see *inet*(3N).

Name2 is the name or address of the gateway to which messages should be forwarded.

Value is a metric indicating the hop count to the destination host or network.

One of the keywords **passive**, **active** or **external** indicates if the gateway should be treated as *passive* or *active* (as described above), or whether the gateway is external to the scope of the *routed* protocol.

Internetwork routers that are directly attached to the Arpanet or Milnet should use the Exterior Gateway Protocol (EGP) to gather routing information rather then using a static routing table of passive gateways. EGP is required in order to provide routes for local networks to the rest of the Internet system. Sites needing assistance with such configurations should contact the Computer Systems Research Group at Berkeley.

FILES

/etc/gateways	for distant gateways

SEE ALSO

udp(7P), htable(1M)

BUGS

The kernel's routing tables may not correspond to those of *routed* when redirects change or add routes. The only remedy for this is to place the routing process in the kernel.

Routed should incorporate other routing protocols, such as EGP. Using separate processes for each requires configuration options to avoid redundant or competing routes.

Routed should listen to intelligent interfaces, such as an IMP, and to error protocols, such as ICMP, to gather more information. It does not always detect unidirectional failures in network interfaces (e.g., when the output side fails).

NAME

rwhod - system status server

SYNOPSIS

/etc/rwhod

DESCRIPTION

Rwhod is the server which maintains the database used by the *rwho*(1C) and *ruptime*(1C) programs. Its operation is predicated on the ability to *broadcast* messages on a network.

Rwhod operates as both a producer and consumer of status information. As a producer of information it periodically queries the state of the system and constructs status messages which are broadcast on a network. As a consumer of information, it listens for other *rwhod* servers' status messages, validating them, then recording them in a collection of files located in the directory */usr/spool/rwho*.

The server transmits and receives messages at the port indicated in the "rwho" service specification; see *services*(5). The messages sent and received, are of the form:

```
struct     outmp {
     char  out_line[8];  /* tty name */
     char  out_name[8];  /* user id */
     long  out_time;     /* time on */
};

struct     whod {
     char  wd_vers;
     char  wd_type;
     char  wd_fill[2];
     int   wd_sendtime;
     int   wd_recvtime;
     char  wd_hostname[32];
     int   wd_loadav[3];
     int   wd_boottime;
     struct whoent {
           struct        outmp we_utmp;
           int           we_idle;
     } wd_we[1024 / sizeof (struct whoent)];
};
```

All fields are converted to network byte order prior to transmission. The load averages are as calculated by the *w*(1) program, and represent load averages over the 5, 10, and 15 minute intervals prior to a server's transmission; they are multiplied by 100 for representation in an integer. The host name included is that returned by the *gethostname*(2) system call, with any trailing domain name omitted. The array at the end of the message contains information about the users logged in to the sending machine. This information includes the contents of the *utmp*(4) entry for each non-idle terminal line and a value indicating the time in seconds since a character was last received on the terminal line.

Messages received by the *rwho* server are discarded unless they originated at an *rwho* server's port. In addition, if the host's name, as specified in the message, contains any unprintable ASCII characters, the message is discarded. Valid messages received by *rwhod* are placed in files named *whod.hostname* in the directory */usr/spool/rwho*. These files contain only the most recent message, in the format described above.

Status messages are generated approximately once every 3 minutes. *Rwhod* performs an *nlist*(3) on /vmunix every 30 minutes to guard against the possibility that this file is not the system image currently operating.

SEE ALSO

rwho(1C), ruptime(1C)

BUGS

There should be a way to relay status information between networks. Status information should be sent only upon request rather than continuously. People often interpret the server dying or network communtication failures as a machine going down.

NAME

sendmail - send mail over the internet

SYNOPSIS

/usr/lib/sendmail [flags] [address ...]

newaliases

mailq [-v]

DESCRIPTION

Sendmail sends a message to one or more *recipients*, routing the message over whatever networks are necessary. *Sendmail* does internetwork forwarding as necessary to deliver the message to the correct place.

Sendmail is not intended as a user interface routine; other programs provide user-friendly front ends; *sendmail* is used only to deliver pre-formatted messages.

With no flags, *sendmail* reads its standard input up to an end-of-file or a line consisting only of a single dot and sends a copy of the message found there to all of the addresses listed. It determines the network(s) to use based on the syntax and contents of the addresses.

Local addresses are looked up in a file and aliased appropriately. Aliasing can be prevented by preceding the address with a backslash. Normally the sender is not included in any alias expansions, e.g., if 'john' sends to 'group', and 'group' includes 'john' in the expansion, then the letter will not be delivered to 'john'.

Flags are:

-ba	Go into ARPANET mode. All input lines must end with a CR-LF, and all messages will be generated with a CR-LF at the end. Also, the ''From:'' and ''Sender:'' fields are examined for the name of the sender.
-bd	Run as a daemon. This requires Berkeley IPC. *Sendmail* will fork and run in background listening on socket 25 for incoming SMTP connections. This is normally run from */etc/rc*.
-bi	Initialize the alias database.
-bm	Deliver mail in the usual way (default).
-bp	Print a listing of the queue.
-bs	Use the SMTP protocol as described in RFC821 on standard input and output. This flag implies all the operations of the **-ba** flag that are compatible with SMTP.
-bt	Run in address test mode. This mode reads addresses and shows the steps in parsing; it is used for debugging configuration tables.
-bv	Verify names only - do not try to collect or deliver a message. Verify mode is normally used for validating users or mailing lists.
-bz	Create the configuration freeze file.
-C*file*	Use alternate configuration file. *Sendmail* refuses to run as root if an alternate configuration file is specified. The frozen configuration file is bypassed.
-d*X*	Set debugging value to *X*.

-F*fullname*	Set the full name of the sender.
-f*name*	Sets the name of the ''from'' person (i.e., the sender of the mail). **-f** can only be used by ''trusted'' users (normally *root, daemon*, and *network)* or if the person you are trying to become is the same as the person you are.
-h*N*	Set the hop count to *N*. The hop count is incremented every time the mail is processed. When it reaches a limit, the mail is returned with an error message, the victim of an aliasing loop. If not specified, ''Received:'' lines in the message are counted.
-n	Don't do aliasing.
-o*x value*	Set option *x* to the specified *value*. Options are described below.
-q[*time*]	Processed saved messages in the queue at given intervals. If *time* is omitted, process the queue once. *Time* is given as a tagged number, with 's' being seconds, 'm' being minutes, 'h' being hours, 'd' being days, and 'w' being weeks. For example, ''-q1h30m'' or ''-q90m'' would both set the timeout to one hour thirty minutes. If *time* is specified, *sendmail* will run in background. This option can be used safely with **-bd**.
-r*name*	An alternate and obsolete form of the **-f** flag.
-t	Read message for recipients. To:, Cc:, and Bcc: lines will be scanned for recipient addresses. The Bcc: line will be deleted before transmission. Any addresses in the argument list will be suppressed, that is, they will *not* receive copies even if listed in the message header.
-v	Go into verbose mode. Alias expansions will be announced, etc.

There are also a number of processing options that may be set. Normally these will only be used by a system administrator. Options may be set either on the command line using the **-o** flag or in the configuration file. These are described in detail in the *Sendmail Installation and Operation Guide.* The options are:

A*file*	Use alternate alias file.
c	On mailers that are considered ''expensive'' to connect to, don't initiate immediate connection. This requires queueing.
d*x*	Set the delivery mode to *x*. Delivery modes are 'i' for interactive (synchronous) delivery, 'b' for background (asynchronous) delivery, and 'q' for queue only - i.e., actual delivery is done the next time the queue is run.
D	Try to automatically rebuild the alias database if necessary.
e*x*	Set error processing to mode *x*. Valid modes are 'm' to mail back the error message, 'w' to ''write'' back the error message (or mail it back if the sender is not logged in), 'p' to print the errors on the terminal (default), 'q' to throw away error messages (only exit status is returned), and 'e' to do special processing for the BerkNet. If the text of the message is not mailed back by modes 'm' or 'w' and if the sender is local to this machine, a copy of the message is appended to the file ''dead.letter'' in the sender's home directory.
F*mode*	The mode to use when creating temporary files.
f	Save UNIX-style From lines at the front of messages.
g*N*	The default group id to use when calling mailers.

H*file*	The SMTP help file.
i	Do not take dots on a line by themselves as a message terminator.
L*n*	The log level.
m	Send to "me" (the sender) also if I am in an alias expansion.
o	If set, this message may have old style headers. If not set, this message is guaranteed to have new style headers (i.e., commas instead of spaces between addresses). If set, an adaptive algorithm is used that will correctly determine the header format in most cases.
Q*queuedir*	Select the directory in which to queue messages.
r*timeout*	The timeout on reads; if none is set, *sendmail* will wait forever for a mailer. This option violates the word (if not the intent) of the SMTP specification, show the timeout should probably be fairly large.
S*file*	Save statistics in the named file.
s	Always instantiate the queue file, even under circumstances where it is not strictly necessary. This provides safety against system crashes during delivery.
T*time*	Set the timeout on undelivered messages in the queue to the specified time. After delivery has failed (e.g., because of a host being down) for this amount of time, failed messages will be returned to the sender. The default is three days.
t*stz,dtz*	Set the name of the time zone.
u*N*	Set the default user id for mailers.

In aliases, the first character of a name may be a vertical bar to cause interpretation of the rest of the name as a command to pipe the mail to. It may be necessary to quote the name to keep *sendmail* from suppressing the blanks from between arguments. For example, a common alias is:

```
        msgs: "|/usr/ucb/msgs -s"
```

Aliases may also have the syntax ":include:*filename*" to ask *sendmail* to read the named file for a list of recipients. For example, an alias such as:

```
        poets: ":include:/usr/local/lib/poets.list"
```

would read */usr/local/lib/poets.list* for the list of addresses making up the group.

Sendmail returns an exit status describing what it did. The codes are defined in <*sysexits.h*>

EX_OK	Successful completion on all addresses.
EX_NOUSER	User name not recognized.
EX_UNAVAILABLE	Catchall meaning necessary resources were not available.
EX_SYNTAX	Syntax error in address.
EX_SOFTWARE	Internal software error, including bad arguments.
EX_OSERR	Temporary operating system error, such as "cannot fork".
EX_NOHOST	Host name not recognized.
EX_TEMPFAIL	Message could not be sent immediately, but was queued.

If invoked as *newaliases, sendmail* will rebuild the alias database. If invoked as *mailq, sendmail* will print the contents of the mail queue.

FILES

Except for /usr/lib/sendmail.cf, these pathnames are all specified in /usr/lib/sendmail.cf. Thus, these values are only approximations.

/usr/lib/aliases	raw data for alias names
/usr/lib/aliases.pag	
/usr/lib/aliases.dir	data base of alias names
/usr/lib/sendmail.cf	configuration file
/usr/lib/sendmail.fc	frozen configuration
/usr/lib/sendmail.hf	help file
/usr/lib/sendmail.st	collected statistics
/usr/spool/mqueue/*	temp files

SEE ALSO

binmail(1), mail(1), rmail(1), syslog(3), aliases(4), sendmail.cf(4), mailaddr(5), rc(8);
DARPA Internet Request For Comments RFC819, RFC821, RFC822;
Sendmail - An Internetwork Mail Router
Sendmail Installation and Operation Guide

NAME

slattach - attach serial lines as network interfaces

SYOPNSIS

/etc/slattach ttyname [*baudrate*]

DESCRIPTION

Slattach is used to assign a tty line to a network interface, and to define the network source and destination addresses. The *ttyname* parameter is a string of the form ''ttyXX'', or ''/dev/ttyXX''. The optional *baudrate* parameter is used to set the speed of the connection. If not specified, the default of 9600 is used.

Only the super-user may attach a network interface.

To detach the interface, use 'ifconfig *interface-name* down' after killing off the *slattach* process. *interface-name* is the name that is shown by **netstat(1)**

EXAMPLES

```
/etc/slattach ttyh8
/etc/slattach /dev/tty01 4800
```

DIAGNOSTICS

Messages indicating the specified interface does not exit, the requested address is unknown, the user is not privileged and tried to alter an interface's configuration.

SEE ALSO

rc(8), intro(7N), netstat(1), ifconfig(1M)

NAME

syslogd - log systems messages

SYNOPSIS

/etc/syslogd [**-f***configfile*] [**-m***markinterval*] [**-d**]

DESCRIPTION

Syslogd reads and logs messages into a set of files described by the configuration file /etc/syslog.conf. Each message is one line. A message can contain a priority code, marked by a number in angle braces at the beginning of the line. Priorities are defined in <*sys/syslog.h*>. *Syslogd* reads from the UNIX domain socket */dev/log*, from an Internet domain socket specified in */etc/services*, and from the special device */dev/klog* (to read kernel messages).

Syslogd configures when it starts up and whenever it receives a hangup signal. Lines in the configuration file have a *selector* to determine the message priorities to which the line applies and an *action*. The *action* field are separated from the selector by one or more tabs.

Selectors are semicolon separated lists of priority specifiers. Each priority has a *facility* describing the part of the system that generated the message, a dot, and a *level* indicating the severity of the message. Symbolic names may be used. An asterisk selects all facilities. All messages of the specified level or higher (greater severity) are selected. More than one facility may be selected using commas to separate them. For example:

```
*.emerg;mail,daemon.crit
```

Selects all facilities at the *emerg* level and the *mail* and *daemon* facilities at the *crit* level.

Known facilities and levels recognized by *syslogd* are those listed in *syslog*(3) without the leading ''LOG_''. The additional facility ''mark'' has a message at priority LOG_INFO sent to it every 20 minutes (this may be changed with the **-m** flag). The ''mark'' facility is not enabled by a facility field containing an asterisk. The level ''none'' may be used to disable a particular facility. For example,

```
*.debug;mail.none
```

Sends all messages *except* mail messages to the selected file.

The second part of each line describes where the message is to be logged if this line is selected. There are four forms:

- A filename (beginning with a leading slash). The file will be opened in append mode.
- A hostname preceeded by an at sign (''@''). Selected messages are forwarded to the *syslogd* on the named host.
- A comma separated list of users. Selected messages are written to those users if they are logged in.
- An asterisk. Selected messages are written to all logged-in users.

Blank lines and lines beginning with '#' are ignored.

For example, the configuration file:

```
kern,mark.debug         /dev/console
*.notice;mail.info      /usr/spool/adm/syslog
*.crit                  /usr/adm/critical
kern.err                @ucbarpa
```

```
*.emerg                     *
*.alert                     eric,kridle
*.alert;auth.warning        ralph
```

logs all kernel messages and 20 minute marks onto the system console, all notice (or higher) level messages and all mail system messages except debug messages into the file /usr/spool/adm/syslog, and all critical messages into /usr/adm/critical; kernel messages of error severity or higher are forwarded to ucbarpa. All users will be informed of any emergency messages, the users ''eric'' and ''kridle'' will be informed of any alert messages, and the user ''ralph'' will be informed of any alert message, or any warning message (or higher) from the authorization system.

The flags are:

-f	Specify an alternate configuration file.
-m	Select the number of minutes between mark messages.
-d	Turn on debugging.

Syslogd creates the file /etc/syslog.pid, if possible, containing a single line with its process id. This can be used to kill or reconfigure *syslogd.*

To bring *syslogd* down, it should be sent a terminate signal (e.g. kill `cat /etc/syslog.pid`).

FILES

/etc/syslog.conf	the configuration file
/etc/syslog.pid	the process id
/dev/log	Name of the UNIX domain datagram log socket
/dev/klog	The kernel log device

SEE ALSO

logger(1), syslog(3)

NAME

talkd - remote user communication server

SYNOPSIS

/etc/talkd

DESCRIPTION

Talkd is the server that notifies a user that somebody else wants to initiate a conversation. It acts a repository of invitations, responding to requests by clients wishing to rendezvous to hold a conversation. In normal operation, a client, the caller, initiates a rendezvous by sending a CTL_MSG to the server of type LOOK_UP (see <*protocols/talkd.h*>). This causes the server to search its invitation tables to check if an invitation currently exists for the caller (to speak to the callee specified in the message). If the lookup fails, the caller then sends an ANNOUNCE message causing the server to broadcast an announcement on the callee's login ports requesting contact. When the callee responds, the local server uses the recorded invitation to respond with the appropriate rendezvous address and the caller and callee client programs establish a stream connection through which the conversation takes place.

SEE ALSO

talk(1), write(1)

NAME

telnetd - DARPA TELNET protocol server

SYNOPSIS

/etc/telnetd

DESCRIPTION

Telnetd is a server which supports the DARPA standard **TELNET** virtual terminal protocol. *Telnetd* is invoked by the internet server (see *inetd*(8)), normally for requests to connect to the **TELNET** port as indicated by the */etc/services* file (see *services*(4)).

Telnetd operates by allocating a pseudo-terminal device (see *pty*(4)) for a client, then creating a login process which has the slave side of the pseudo-terminal as **stdin**, **stdout**, and **stderr**. *Telnetd* manipulates the master side of the pseudo-terminal, implementing the **TELNET** protocol and passing characters between the remote client and the login process.

When a **TELNET** session is started up, *telnetd* sends **TELNET** options to the client side indicating a willingness to do *remote echo* of characters, to *suppress go ahead*, and to receive *terminal type information* from the remote client. If the remote client is willing, the remote terminal type is propagated in the environment of the created login process.

Telnetd is willing to *do*: *echo*, *binary*, *suppress go ahead*, and *timing mark*. *Telnetd* is willing to have the remote client *do*: *binary*, *terminal type*, and *suppress go ahead*.

SEE ALSO

telnet(1C)

BUGS

Some **TELNET** commands are only partially implemented.

The **TELNET** protocol allows for the exchange of the number of lines and columns on the user's terminal, but *telnetd* doesn't make use of them.

Because of bugs in the original 4.2 BSD *telnet*(1C), *telnetd* performs some dubious protocol exchanges to try to discover if the remote client is, in fact, a 4.2 BSD *telnet*(1C).

Binary mode has no common interpretation except between similar operating systems (Unix in this case).

The terminal type name received from the remote client is converted to lower case.

The *packet* interface to the pseudo-terminal (see *pty*(4)) should be used for more intelligent flushing of input and output queues.

Telnetd never sends **TELNET** *go ahead* commands.

NAME

tftpd - DARPA Trivial File Transfer Protocol server

SYNOPSIS

/etc/tftpd

DESCRIPTION

Tftpd is a server which supports the DARPA Trivial File Transfer Protocol. The TFTP server operates at the port indicated in the ''tftp'' service description; see *services*(4). The server is normally started by *inetd*(1M).

The use of *tftp* does not require an account or password on the remote system. Due to the lack of authentication information, *tftpd* will allow only publicly readable files to be accessed. Files may be written only if they already exist and are publicly writable. Note that this extends the concept of ''public'' to include all users on all hosts that can be reached through the network; this may not be appropriate on all systems, and its implications should be considered before enabling tftp service. The server should have the user ID with the lowest possible privilege.

SEE ALSO

tftp(1C), inetd(1M)

NAME

trpt - transliterate protocol trace

SYNOPSIS

trpt [**-a**] [**-s**] [**-t**] [**-f**] [**-j**] [**-p** hex-address] [system [core]]

DESCRIPTION

Trpt interrogates the buffer of TCP trace records created when a socket is marked for "debugging" (see *setsockopt*(2)), and prints a readable description of these records. When no options are supplied, *trpt* prints all the trace records found in the system grouped according to TCP connection protocol control block (PCB). The following options may be used to alter this behavior.

-a	in addition to the normal output, print the values of the source and destination addresses for each packet recorded.
-s	in addition to the normal output, print a detailed description of the packet sequencing information.
-t	in addition to the normal output, print the values for all timers at each point in the trace.
-f	follow the trace as it occurs, waiting a short time for additional records each time the end of the log is reached.
-j	just give a list of the protocol control block addresses for which there are trace records.
-p	show only trace records associated with the protocol control block, the address of which follows.

The recommended use of *trpt* is as follows. Isolate the problem and enable debugging on the socket(s) involved in the connection. Find the address of the protocol control blocks associated with the sockets using the **-A** option to *netstat*(1). Then run *trpt* with the **-p** option, supplying the associated protocol control block addresses. The **-f** option can be used to follow the trace log once the trace is located. If there are many sockets using the debugging option, the **-j** option may be useful in checking to see if any trace records are present for the socket in question.

If debugging is being performed on a system or core file other than the default, the last two arguments may be used to supplant the defaults.

FILES

/vmunix
/dev/kmem

SEE ALSO

setsockopt(2), netstat(1), trsp(1M)

DIAGNOSTICS

"no namelist" when the system image doesn't contain the proper symbols to find the trace buffer; others which should be self explanatory.

BUGS

Should also print the data for each input or output, but this is not saved in the race record.

The output format is inscrutable and should be described here.

NAME

accept - accept a connection on a socket

SYNOPSIS

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

ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr *addr;
int *addrlen;

DESCRIPTION

The argument *s* is a socket that has been created with *socket*(2), bound to an address with *bind*(2), and is listening for connections after a *listen*(2). *Accept* extracts the first connection on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor, *ns*, for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *accept* blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, *accept* returns an error as described below. The accepted socket, *ns*, may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with SOCK_STREAM.

It is possible to *select*(2) a socket for the purposes of doing an *accept* by selecting it for read.

RETURN VALUE

The call returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

ERRORS

The *accept* will fail if:

[EBADF]	The descriptor is invalid.
[ENOTSOCK]	The descriptor references a file, not a socket.
[EOPNOTSUPP]	The referenced socket is not of type SOCK_STREAM.
[EFAULT]	The *addr* parameter is not in a writable part of the user address space.
[EWOULDBLOCK]	The socket is marked non-blocking and no connections are present to be accepted.

SEE ALSO

bind(2), connect(2), listen(2), select(2), socket(2)

NAME
bind - bind a name to a socket

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

bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;

DESCRIPTION
Bind assigns a name to an unnamed socket. When a socket is created with *socket*(2) it exists in a name space (address family) but has no name assigned. *Bind* requests that *name* be assigned to the socket.

NOTES
Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed (using *unlink*(2)).

The rules used in name binding vary between communication domains. Consult the manual entries for Networking Protocols in Appendix A for detailed information.

RETURN VALUE
If the bind is successful, a 0 value is returned. A return value of -1 indicates an error, which is further specified in the global *errno*.

ERRORS
The *bind* call will fail if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is not a socket.
[EADDRNOTAVAIL]	The specified address is not available from the local machine.
[EADDRINUSE]	The specified address is already in use.
[EINVAL]	The socket is already bound to an address.
[EACCES]	The requested address is protected, and the current user has inadequate permission to access it.
[EFAULT]	The *name* parameter is not in a valid part of the user address space.

The following errors are specific to binding names in the UNIX domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	A prefix component of the path name does not exist.

[ELOOP]	Too many symbolic links were encountered in translating the pathname.
[EIO]	An I/O error occurred while making the directory entry or allocating the inode.
[EROFS]	The name would reside on a read-only file system.
[EISDIR]	A null pathname was specified.

SEE ALSO

connect(2), listen(2), socket(2), getsockname(2)

NAME

connect - initiate a connection on a socket

SYNOPSIS

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

connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;

DESCRIPTION

The parameter *s* is a socket. If it is of type SOCK_DGRAM, then this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type SOCK_STREAM, then this call attempts to make a connection to another socket. The other socket is specified by *name*, which is an address in the communications space of the socket. Each communications space interprets the *name* parameter in its own way. Generally, stream sockets may successfully *connect* only once; datagram sockets may use *connect* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address.

RETURN VALUE

If the connection or binding succeeds, then 0 is returned. Otherwise a -1 is returned, and a more specific error code is stored in *errno*.

ERRORS

The call fails if:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a descriptor for a file, not a socket.
[EADDRNOTAVAIL]	The specified address is not available on this machine.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EISCONN]	The socket is already connected.
[ETIMEDOUT]	Connection establishment timed out without establishing a connection.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ENETUNREACH]	The network isn't reachable from this host.
[EADDRINUSE]	The address is already in use.
[EFAULT]	The *name* parameter specifies an area outside the process address space.
[EINPROGRESS]	The socket is non-blocking and the connection cannot be completed immediately. It is possible to *select*(2) for completion by selecting the socket for writing.

[EALREADY]	The socket is non-blocking and a previous connection attempt has not yet been completed.

The following errors are specific to connecting names in the UNIX domain. These errors may not apply in future versions of the UNIX IPC domain.

[ENOTDIR]	A component of the path prefix is not a directory.
[EINVAL]	The pathname contains a character with the high-order bit set.
[ENAMETOOLONG]	A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOENT]	The named socket does not exist.
[EACCES]	Search permission is denied for a component of the path prefix.
[EACCES]	Write access to the named socket is denied.
[ELOOP]	Too many symbolic links were encountered in translating the pathname.

SEE ALSO

accept(2), select(2), socket(2), getsockname(2)

NAME

fchmod - change mode of file

SYNOPSIS

int fchmod (fildes, mode)
int fildes;
int mode;

DESCRIPTION

Fildes is a file descriptor for an open file that may have been returned from a *open*(2) or *dup*(2) call. *Fchmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode.*

Access permission bits are interpreted as follows:

center; 1 1 1 l. 04000 Set user ID on execution. 02000 Set group ID on execution. 01000 Save text image after execution. 00400 Read by owner. 00200 Write by owner. 00100 Execute (search if a directory) by owner. 00070 Read, write, execute (search) by group. 00007 Read, write, execute (search) by others.

The effective user ID of the process must match the owner of the file or be super-user to change the mode of a file.

If the effective user ID of the process is not super-user, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not super-user and the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If an executable file is prepared for sharing then mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus, when the next user of the file executes it, the text need not be read from the file system but can simply be swapped in, saving time.

Fchmod will fail and the file mode will be unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), chown(2), fchown(2), mknod(2).

NAME

fchown - change owner and group of a file

SYNOPSIS

int fchown (fildes, owner, group)
int fildes;
int owner, group;

DESCRIPTION

Fildes is a file descriptor for an open file that may have been returned from a *open*(2) or *dup*(2) call. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with effective user ID equal to the file owner or super-user may change the ownership of a file.

If *fchown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

Fchown will fail and the owner and group of the named file will remain unchanged if one or more of the following are true:

[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	The named file does not exist.
[EACCES]	Search permission is denied on a component of the path prefix.
[EPERM]	The effective user ID does not match the owner of the file and the effective user ID is not super-user.
[EROFS]	The named file resides on a read-only file system.
[EFAULT]	*Path* points outside the allocated address space of the process.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

chmod(2), chown(2), fchmod(2).
chown(1) in the *ICON/UXV User Reference Manual.*

NAME

gethostid, sethostid - get/set unique identifier of current host

SYNOPSIS

hostid = gethostid()
long hostid;

sethostid(hostid)
long hostid;

DESCRIPTION

Sethostid establishes a 32-bit identifier for the current processor that is intended to be unique among all UNIX systems in existence. This is normally a DARPA Internet address for the local machine. This call is allowed only to the super-user and is normally performed at boot time.

Gethostid returns the 32-bit identifier for the current processor.

SEE ALSO

gethostname(2)

NAME

gethostname, sethostname - get/set name of current host

SYNOPSIS

gethostname(name, namelen)
char *name;
int namelen;

sethostname(name, namelen)
char *name;
int namelen;

DESCRIPTION

Gethostname returns the standard host name for the current processor, as previously set by *sethostname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

Sethostname sets the name of the host machine to be *name*, which has length *namelen*. This call is restricted to the super-user and is normally used only when the system is bootstrapped.

RETURN VALUE

If the call succeeds a value of 0 is returned. If the call fails, then a value of -1 is returned and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* or *namelen* parameter gave an invalid address.
[EPERM]	The caller tried to set the hostname and was not the super-user.

SEE ALSO

gethostid(2)

BUGS

Host names are limited to MAXHOSTNAMELEN (from <*sys/param.h*>) characters, currently 64.

NAME

getpeername - get name of connected peer

SYNOPSIS

getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;

DESCRIPTION

Getpeername returns the name of the peer connected to socket *s*. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes). The name is truncated if the buffer provided is too small.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOTCONN]	The socket is not connected.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

accept(2), bind(2), socket(2), getsockname(2)

NAME

getsockname - get socket name

SYNOPSIS

getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;

DESCRIPTION

Getsockname returns the current *name* for the specified socket. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *name*. On return it contains the actual size of the name returned (in bytes).

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOBUFS]	Insufficient resources were available in the system to perform the operation.
[EFAULT]	The *name* parameter points to memory not in a valid part of the process address space.

SEE ALSO

bind(2), socket(2)

BUGS

Names bound to sockets in the UNIX domain are inaccessible; *getsockname* returns a zero length name.

NAME

getsockopt, setsockopt - get and set options on sockets

SYNOPSIS

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

getsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int *optlen;

setsockopt(s, level, optname, optval, optlen)
int s, level, optname;
char *optval;
int optlen;
```

DESCRIPTION

Getsockopt and *setsockopt* manipulate *options* associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost ''socket'' level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the ''socket'' level, *level* is specified as SOL_SOCKET. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, *level* should be set to the protocol number of TCP; see *getprotoent*(3N).

The parameters *optval* and *optlen* are used to access option values for *setsockopt*. For *getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be supplied as 0.

Optname and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file <*sys/socket.h*> contains definitions for ''socket'' level options, described below. Options at other protocol levels vary in format and name; consult the appropriate manual entries for section 7P in Appendix A.

Most socket-level options take an *int* parameter for *optval*. For *setsockopt*, the parameter should non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a *struct linger* parameter, defined in <*sys/socket.h*>, which specifies the desired state of the option and the linger interval (see below).

The following options are recognized at the socket level. Except as noted, each may be examined with *getsockopt* and set with *setsockopt*.

SO_DEBUG	toggle recording of debugging information
SO_REUSEADDR	toggle local address reuse
SO_KEEPALIVE	toggle keep connections alive
SO_DONTROUTE	toggle routing bypass for outgoing messages
SO_LINGER	linger on close if data present

SO_BROADCAST	toggle permission to transmit broadcast messages
SO_OOBINLINE	toggle reception of out-of-band data in band
SO_SNDBUF	set buffer size for output
SO_RCVBUF	set buffer size for input
SO_TYPE	get the type of the socket (get only)
SO_ERROR	get and clear error on the socket (get only)

SO_DEBUG enables debugging in the underlying protocol modules. SO_REUSEADDR indicates that the rules used in validating addresses supplied in a *bind*(2) call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via a SIGPIPE signal. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_LINGER controls the action taken when unsent messags are queued on socket and a *close*(2) is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the *close* attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *setsockopt* call when SO_LINGER is requested). If SO_LINGER is disabled and a *close* is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the system. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *recv* or *read* calls without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values. Finally, SO_TYPE and SO_ERROR are options used only with *setsockopt*. SO_TYPE returns the type of the socket, such as SOCK_STREAM; it is useful for servers that inherit sockets on startup. SO_ERROR returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

RETURN VALUE

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is a file, not a socket.
[ENOPROTOOPT]	The option is unknown at the level indicated.
[EFAULT]	The address pointed to by *optval* is not in a valid part of the process address space. For *getsockopt*, this error may also be returned if *optlen* is not in a valid part of the process address space.

SEE ALSO

ioctl(2), socket(2), getprotoent(3N)

BUGS

Several of the socket options should be handled at lower levels of the system.

NAME

gettimeofday, settimeofday - get/set date and time

SYNOPSIS

#include <sys/time.h>

gettimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;

settimeofday(tp, tzp)
struct timeval *tp;
struct timezone *tzp;

DESCRIPTION

The system's notion of the current Greenwich time and the current time zone is obtained with the *gettimeofday* call, and set with the *settimeofday* call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent, and the time may be updated continuously or in ''ticks.'' If *tzp* is zero, the time zone information will not be returned or set.

The structures pointed to by *tp* and *tzp* are defined in <*sys/time.h*> as:

```
struct timeval {
        long    tv_sec;         /* seconds since Jan. 1, 1970 */
        long    tv_usec;                /* and microseconds */
};

struct timezone {
        int     tz_minuteswest;         /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply */
};
```

The *timezone* structure indicates the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

Only the super-user may set the time of day or time zone.

RETURN

A 0 return value indicates that the call succeeded. A -1 return value indicates an error occurred, and in this case an error code is stored into the global variable *errno*.

ERRORS

The following error codes may be set in *errno*:

[EFAULT]	An argument address referenced invalid memory.
[EPERM]	A user other than the super-user attempted to set the time.

SEE ALSO

date(1), ctime(3)

NAME

listen - listen for connections on a socket

SYNOPSIS

listen(s, backlog)
int s, backlog;

DESCRIPTION

To accept connections, a socket is first created with *socket*(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with *listen*(2), and then the connections are accepted with *accept*(2). The *listen* call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.

The *backlog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

RETURN VALUE

A 0 return value indicates success; -1 indicates an error.

ERRORS

The call fails if:

[EBADF]	The argument *s* is not a valid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EOPNOTSUPP]	The socket is not of a type that supports the operation *listen*.

SEE ALSO

accept(2), connect(2), socket(2)

BUGS

The *backlog* is currently limited (silently) to 5.

NAME

readv - read input

SYNOPSIS

#include <sys/types.h>
#include <sys/uio.h>

cc = readv(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;

DESCRIPTION

Readv performs the same action, but scatters the input data into the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt - 1].

For *readv*, the *iovec* structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *Readv* will always fill an area completely before proceeding to the next.

On objects capable of seeking, the read starts at a position given by the pointer associated with *d* (see *lseek*(2)). Upon return from *read*, the pointer is incremented by the number of bytes actually read.

Objects that are not capable of seeking always read from the current position. The value of the pointer associated with such an object is undefined.

Upon successful completion, *readv* returns the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a normal file that has that many bytes left before the end-of-file, but in no other case.

If the returned value is 0, then end-of-file has been reached.

RETURN VALUE

If successful, the number of bytes actually read is returned. Otherwise, a -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

Readv will fail if one or more of the following are true:

[EBADF]	*D* is not a valid file or socket descriptor open for reading.
[EIO]	An I/O error occurred while reading from the file system.
[EINTR]	A read from a slow device was interrupted before any data arrived by the delivery of a signal.
[EINVAL]	The pointer associated with *d* was negative.

[EWOULDBLOCK]	The file was marked for non-blocking I/O, and no data were ready to be read.

In addition, *readv* may return one of the following errors:

[EINVAL]	*Iovcnt* was less than or equal to 0, or greater than 16.
[EINVAL]	One of the *iov_len* values in the *iov* array was negative.
[EINVAL]	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.
[EFAULT]	Part of the *iov* points outside the process's allocated address space.

SEE ALSO

dup(2), fcntl(2), open(2), pipe(2), select(2), socket(2), socketpair(2)

NAME

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS

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

cc = recv(s, buf, len, flags)
int cc, s;
char *buf;
int len, flags;

cc = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;

cc = recvmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;

DESCRIPTION

Recv, *recvfrom*, and *recvmsg* are used to receive messages from a socket.

The *recv* call is normally used only on a *connected* socket (see *connect*(2)), while *recvfrom* and *recvmsg* may be used to receive data on a socket whether it is in a connected state or not.

If *from* is non-zero, the source address of the message is filled in. *Fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned in *cc*. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see *socket*(2)).

If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is nonblocking (see *ioctl*(2)) in which case a *cc* of -1 is returned with the external variable errno set to EWOULDBLOCK.

The *select*(2) call may be used to determine when more data arrives.

The *flags* argument to a recv call is formed by *or*'ing one or more of the values,

```
#define    MSG_OOB     0x1   /* process out-of-band data */
#define    MSG_PEEK    0x2   /* peek at incoming message */
```

The *recvmsg* call uses a *msghdr* structure to minimize the number of directly supplied parameters. This structure has the following form, as defined in *<sys/socket.h>*:

```
struct msghdr {
     caddr_t  msg_name;        /* optional address */
     int      msg_namelen;     /* size of address */
     struct   iovec *msg_iov;  /* scatter/gather array */
     int      msg_iovlen;      /* # elements in msg_iov */
     caddr_t  msg_accrights;   /* access rights sent/received */
     int      msg_accrightslen;
};
```

Here *msg_name* and *msg_namelen* specify the destination address if the socket is unconnected; *msg_name* may be given as a null pointer if no names are desired or required. The *msg_iov* and *msg_iovlen* describe the scatter gather locations, as described in *readv*(2). A buffer to receive any access rights sent along with the message is specified in *msg_accrights*, which has length *msg_accrightslen*. Access rights are currently limited to file descriptors, which each occupy the size of an **int**.

RETURN VALUE

These calls return the number of bytes received, or -1 if an error occurred.

ERRORS

The calls fail if:

[EBADF]	The argument *s* is an invalid descriptor.
[ENOTSOCK]	The argument *s* is not a socket.
[EWOULDBLOCK]	The socket is marked non-blocking and the receive operation would block.
[EINTR]	The receive was interrupted by delivery of a signal before any data was available for the receive.
[EFAULT]	The data was specified to be received into a non-existent or protected part of the process address space.

SEE ALSO

fcntl(2), read(2), send(2), select(2), getsockopt(2), socket(2)

NAME

select - synchronous I/O multiplexing

SYNOPSIS

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

nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;

FD_SET(fd, &fdset)
FD_CLR(fd, &fdset)
FD_ISSET(fd, &fdset)
FD_ZERO(&fdset)
int fd;
fd_set fdset;
```

DESCRIPTION

Select examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfds* descriptors are checked in each set; i.e. the descriptors from 0 through *nfds*-1 in the descriptor sets are examined. On return, *select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. The total number of ready descriptors in all the sets is returned in *nfound*.

The descriptor sets are stored as bit fields in arrays of integers. The following macros are provided for manipulating such descriptor sets: *FD_ZERO(&fdset)* initializes a descriptor set *fdset* to the null set. *FD_SET(fd, &fdset)* includes a particular descriptor *fd* in *fdset*. *FD_CLR(fd, &fdset)* removes *fd* from *fdset*. *FD_ISSET(fd, &fdset)* is nonzero if *fd* is a member of *fdset*, zero otherwise. The behavior of these macros is undefined if a descriptor value is less than zero or greater than or equal to *FD_SETSIZE*, which is normally at least equal to the maximum number of descriptors supported by the system.

If *timeout* is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a zero pointer, the select blocks indefinitely. To affect a poll, the *timeout* argument should be non-zero, pointing to a zero-valued timeval structure.

Any of *readfds*, *writefds*, and *exceptfds* may be given as zero pointers if no descriptors are of interest.

RETURN VALUE

Select returns the number of ready descriptors that are contained in the descriptor sets, or -1 if an error occurred. If the time limit expires then *select* returns 0. If *select* returns with an error, including one due to an interrupted call, the descriptor sets will be unmodified.

ERRORS

An error return from *select* indicates:

[EBADF]	One of the descriptor sets specified an invalid descriptor.
[EINTR]	A signal was delivered before the time limit expired and before any of the selected events occurred.
[EINVAL]	The specified time limit is invalid. One of its components is negative or too large.

SEE ALSO

accept(2), connect(2), read(2), write(2), recv(2), send(2)

NAME

send, sendto, sendmsg - send a message from a socket

SYNOPSIS

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

cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;

cc = sendto(s, msg, len, flags, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr *to;
int tolen;

cc = sendmsg(s, msg, flags)
int cc, s;
struct msghdr msg[];
int flags;

DESCRIPTION

Send, *sendto*, and *sendmsg* are used to transmit a message to another socket. *Send* may be used only when the socket is in a *connected* state, while *sendto* and *sendmsg* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, then the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a *send*. Return values of -1 indicate some locally detected errors.

If no messages space is available at the socket to hold the message to be transmitted, then *send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The *select*(2) call may be used to determine when it is possible to send more data.

The *flags* parameter may include one or more of the following:

```
#define    MSG_OOB         0x1  /* process out-of-band data */
#define    MSG_DONTROUTE   0x4  /* bypass routing, use direct interface */
```

The flag MSG_OOB is used to send "out-of-band" data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support "out-of-band" data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.

See *recv*(2) for a description of the *msghdr* structure.

RETURN VALUE

The call returns the number of characters sent, or -1 if an error occurred.

ERRORS

[EBADF]	An invalid descriptor was specified.
[ENOTSOCK]	The argument *s* is not a socket.
[EFAULT]	An invalid user space address was specified for a parameter.
[EMSGSIZE]	The socket requires that message be sent atomically, and the size of the message to be sent made this impossible.
[EWOULDBLOCK]	The socket is marked non-blocking and the requested operation would block.
[ENOBUFS]	The system was unable to allocate an internal buffer. The operation may succeed when buffers become available.
[ENOBUFS]	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.

SEE ALSO

fcntl(2), recv(2), select(2), getsockopt(2), socket(2), write(2)

NAME

shutdown - shut down part of a full-duplex connection

SYNOPSIS

shutdown(s, how)
int s, how;

DESCRIPTION

The *shutdown* call causes all or part of a full-duplex connection on the socket associated with *s* to be shut down. If *how* is 0, then further receives will be disallowed. If *how* is 1, then further sends will be disallowed. If *how* is 2, then further sends and receives will be disallowed.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EBADF]	*S* is not a valid descriptor.
[ENOTSOCK]	*S* is a file, not a socket.
[ENOTCONN]	The specified socket is not connected.

SEE ALSO

connect(2), socket(2)

NAME

socket - create an endpoint for communication

SYNOPSIS

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

s = socket(domain, type, protocol)
int s, domain, type, protocol;

DESCRIPTION

Socket creates an endpoint for communication and returns a descriptor.

The *domain* parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. The protocol family generally is the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file <*sys/socket.h*>. The currently understood formats are

PF_UNIX	(UNIX internal protocols),
PF_INET	(ARPA Internet protocols),

The socket has the indicated *type,* which specifies the semantics of communication. Currently defined types are:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

A SOCK_STREAM type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length). SOCK_RAW sockets provide access to internal network protocols and interfaces. The types SOCK_RAW, which is available only to the super-user, is not described here.

The *protocol* specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the ``communication domain'' in which communication is to take place.

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a *connected* state before any data may be sent or received on it. A connection to another socket is created with a *connect*(2) call. Once connected, data may be transferred using *read*(2) and *write*(2) calls or some variant of the *send*(2) and *recv*(2) calls. When a session has been completed a *close*(2) may be performed. Out-of-band data may also be transmitted as described in *send*(2) and received as described in *recv*(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with -1 returns and with ETIMEDOUT as the specific code in the global variable errno. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for a extended period (e.g. 5 minutes). A SIGPIPE signal is raised if a process sends on a broken stream; this causes naive processes, which do not handle the signal, to exit.

SOCK_DGRAM and SOCK_RAW sockets allow sending of datagrams to correspondents named in *send*(2) calls. Datagrams are generally received with *recvfrom*(2), which returns the next datagram with its return address.

The operation of sockets is controlled by socket level *options*. These options are defined in the file <*sys/socket.h*>. *Setsockopt*(2) and *getsockopt*(2) are used to set and get options, respectively.

RETURN VALUE

A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

ERRORS

The *socket* call fails if:

[EPROTONOSUPPORT]	The protocol type or the specified protocol is not supported within this domain.
[EMFILE]	The per-process descriptor table is full.
[ENFILE]	The system file table is full.
[EACCESS]	Permission to create a socket of the specified type and/or protocol is denied.
[ENOBUFS]	Insufficient buffer space is available. The socket cannot be created until sufficient resources are freed.

SEE ALSO

accept(2), bind(2), connect(2), getsockname(2), getsockopt(2), ioctl(2), listen(2), read(2), recv(2), select(2), send(2), shutdown(2), socketpair(2), write(2)
"Appendix F: An Introductory 4.3BSD Interprocess Communication Tutorial." "Appendix G: An Advanced 4.3BSD Interprocess Communication Tutorial."

NAME

socketpair - create a pair of connected sockets

SYNOPSIS

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

socketpair(d, type, protocol, sv)
int d, type, protocol;
int sv[2];

DESCRIPTION

The *socketpair* call creates an unnamed pair of connected sockets in the specified domain *d*, of the specified *type*, and using the optionally specified *protocol*. The descriptors used in referencing the new sockets are returned in *sv*[0] and *sv*[1]. The two sockets are indistinguishable.

DIAGNOSTICS

A 0 is returned if the call succeeds, -1 if it fails.

ERRORS

The call succeeds unless:

[EMFILE]	Too many descriptors are in use by this process.
[EAFNOSUPPORT]	The specified address family is not supported on this machine.
[EPROTONOSUPPORT]	The specified protocol is not supported on this machine.
[EOPNOSUPPORT]	The specified protocol does not support creation of socket pairs.
[EFAULT]	The address *sv* does not specify a valid part of the process address space.

SEE ALSO

read(2), write(2), pipe(2)

BUGS

This call is currently implemented only for the UNIX domain.

NAME

vfork - spawn new process in a virtual memory efficient way

SYNOPSIS

pid = vfork()
int pid;

DESCRIPTION

Vfork can be used to create new processes without fully copying the address space of the old process, which is horrendously inefficient in a paged environment. It is useful when the purpose of *fork*(2) would have been to create a new system context for an *execve*. *Vfork* differs from *fork* in that the child borrows the parent's memory and thread of control until a call to *execve*(2) or an exit (either by a call to *exit*(2) or abnormally.) The parent process is suspended while the child is using its resources.

Vfork returns 0 in the child's context and (later) the pid of the child in the parent's context.

Vfork can normally be used just like *fork*. It does not work, however, to return while running in the childs context from the procedure that called *vfork* since the eventual return from *vfork* would then return to a no longer existent stack frame. Be careful, also, to call *_exit* rather than *exit* if you can't *execve*, since *exit* will flush and close standard I/O channels, and thereby mess up the parent processes standard I/O data structures. (Even with *fork* it is wrong to call *exit* since buffered data would then be flushed twice.)

SEE ALSO

fork(2), execve(2), sigvec(2), wait(2),

DIAGNOSTICS

Same as for *fork*.

BUGS

This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of *vfork* as it will, in that case, be made synonymous to *fork*.

To avoid a possible deadlock situation, processes that are children in the middle of a *vfork* are never sent SIGTTOU or SIGTTIN signals; rather, output or *ioctl*s are allowed and input attempts result in an end-of-file indication.

NAME

writev - write output

SYNOPSIS

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

cc = writev(d, iov, iovcnt)
int cc, d;
struct iovec *iov;
int iovcnt;
```

DESCRIPTION

Writev performs the same action, but gathers the output data from the *iovcnt* buffers specified by the members of the *iov* array: iov[0], iov[1], ..., iov[iovcnt - 1].

For *writev*, the *iovec* structure is defined as

```
struct iovec {
        caddr_t iov_base;
        int     iov_len;
};
```

Each *iovec* entry specifies the base address and length of an area in memory from which data should be written. *Writev* will always write a complete area before proceeding to the next.

On objects capable of seeking, the write starts at a position given by the pointer associated with *d*, see *lseek*(2). Upon return from write, the pointer is incremented by the number of bytes actually written.

Objects that are not capable of seeking always write from the current position. The value of the pointer associated with such an object is undefined.

If the real user is not the super-user, then *write* clears the set-user-id bit on a file. This prevents penetration of system security by a user who ''captures'' a writable set-user-id file owned by the super-user.

When using non-blocking I/O on objects such as sockets that are subject to flow control, *write* and *writev* may write fewer bytes than requested; the return value must be noted, and the remainder of the operation should be retried when possible.

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise a -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

Write and *writev* will fail and the file pointer will remain unchanged if one or more of the following are true:

[EBADF]	*D* is not a valid descriptor open for writing.
[EPIPE]	An attempt is made to write to a pipe that is not open for reading by any process.
[EPIPE]	An attempt is made to write to a socket of type SOCK_STREAM that is

	not connected to a peer socket.
[EFBIG]	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
[EFAULT]	Part of *iov* or data to be written to the file points outside the process's allocated address space.
[EINVAL]	The pointer associated with *d* was negative.
[ENOSPC]	There is no free space remaining on the file system containing the file.
[EDQUOT]	The user's quota of disk blocks on the file system containing the file has been exhausted.
[EIO]	An I/O error occurred while reading from or writing to the file system.
[EWOULDBLOCK]	The file was marked for non-blocking I/O, and no data could be written immediately.

In addition, *writev* may return one of the following errors:

[EINVAL]	*Iovcnt* was less than or equal to 0, or greater than 16.
[EINVAL]	One of the *iov_len* values in the *iov* array was negative.
[EINVAL]	The sum of the *iov_len* values in the *iov* array overflowed a 32-bit integer.

SEE ALSO

fcntl(2), lseek(2), open(2), pipe(2), select(2)

NAME

rcmd, rresvport, ruserok - routines for returning a stream to a remote command

SYNOPSIS

rem = rcmd(ahost, inport, locuser, remuser, cmd, fd2p);
char **ahost;
int inport;
char *locuser, *remuser, *cmd;
int *fd2p;

s = rresvport(port);
int *port;

ruserok(rhost, superuser, ruser, luser);
char *rhost;
int superuser;
char *ruser, *luser;

DESCRIPTION

Rcmd is a routine used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. *Rresvport* is a routine which returns a descriptor to a socket with an address in the privileged port space. *Ruserok* is a routine used by servers to authenticate clients requesting service with *rcmd*. All three functions are present in the same file and are used by the *rshd*(1M) server (among others).

Rcmd looks up the host **ahost* using *gethostbyname*(3N), returning -1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host and a connection is established to a server residing at the well-known Internet port *inport*.

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be set up, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

The protocol is described in detail in *rshd*(1M).

The *rresvport* routine is used to obtain a socket with a privileged address bound to it. This socket is suitable for use by *rcmd* and several other routines. Privileged Internet ports are those in the range 0 to 1023. Only the super-user is allowed to bind an address of this sort to a socket.

Ruserok takes a remote host's name, as returned by a *gethostbyaddr*(3N) routine, two user names and a flag indicating whether the local user's name is that of the super-user. It then checks the files */etc/hosts.equiv* and, possibly, *.rhosts* in the current working directory (normally the local user's home directory) to see if the request for service is allowed. A 0 is returned if the machine name is listed in the ''hosts.equiv'' file, or the host and remote user name are found in the ''.rhosts'' file; otherwise *ruserok* returns -1. If the *superuser* flag is 1, the checking of the ''host.equiv'' file is bypassed. If the local domain (as obtained from *gethostname* (2)) is the same as the remote domain, only the machine name need be specified.

SEE ALSO

rlogin(1C), rsh(1C), intro(2), rexec(3), rexecd(1M), rlogind(1M), rshd(1M)

DIAGNOSTICS

Rcmd returns a valid socket descriptor on success. It returns -1 on error and prints a diagnostic message on the standard error.

Rresvport returns a valid, bound socket descriptor on success. It returns -1 on error with the global value *errno* set according to the reason for failure. The error code EAGAIN is overloaded to mean ''All network ports in use.''

NAME

rexec - return stream to a remote command

SYNOPSIS

rem = rexec(ahost, inport, user, passwd, cmd, fd2p);
char **ahost;
int inport;
char *user, *passwd, *cmd;
int *fd2p;

DESCRIPTION

Rexec looks up the host **ahost* using *gethostbyname*(3N), returning -1 if the host does not exist. Otherwise **ahost* is set to the standard name of the host. If a username and password are both specified, then these are used to authenticate to the foreign host; otherwise the environment and then the user's *.netrc* file in his home directory are searched for appropriate information. If all this fails, the user is prompted for the information.

The port *inport* specifies which well-known DARPA Internet port to use for the connection; the call ``getservbyname("exec", "tcp")'' (see *getservent*(3N)) will return a pointer to a structure, which contains the necessary port. The protocol for connection is described in detail in *rexecd*(1M).

If the connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the caller, and given to the remote command as **stdin** and **stdout**. If *fd2p* is non-zero, then an auxiliary channel to a control process will be setup, and a descriptor for it will be placed in **fd2p*. The control process will return diagnostic output from the command (unit 2) on this channel, and will also accept bytes on this channel as being UNIX signal numbers, to be forwarded to the process group of the command. The diagnostic information returned does not include remote authorization failure, as the secondary connection is set up after authorization has been verified. If *fd2p* is 0, then the **stderr** (unit 2 of the remote command) will be made the same as the **stdout** and no provision is made for sending arbitrary signals to the remote process, although you may be able to get its attention by using out-of-band data.

SEE ALSO

rcmd(3), rexecd(1M)

NAME

intro — introduction to the networking library

SYNOPSIS

cc [flags] file [...] -lnet [...]

DESCRIPTION

The networking library contains the following facilities:

Routine	Man Page	Description
endhostent	GETHOSTBYNAME (3N)	end network host entry
endnetent	GETNETENT (3N)	end network entry
endprotoent	GETPROTOENT (3N)	end protocol entry
endservent	GETSERVENT (3N)	end service entry
htonl	BYTEORDER (3N)	convert host and network byte order values
htons	BYTEORDER (3N)	convert host and network byte order values
gethostbyaddr	GETHOSTBYNAME (3N)	get network host entry by address
gethostbyname	GETHOSTBYNAME (3N)	get network host entry by name
gethostent	GETHOSTBYNAME (3N)	get network host entry
getnetbyaddr	GETNETENT (3N)	get network entry by address
getnetbyname	GETNETENT (3N)	get network entry by name
getnetent	GETNETENT (3N)	get network entry
getprotobyname	GETPROTOENT (3N)	get protocol entry by name
getprotobynumber	GETPROTOENT (3N)	get protocol entry by number
getprotoent	GETPROTOENT (3N)	get protocol entry
getservbyname	GETSERVENT (3N)	get service entry by name
getservbyport	GETSERVENT (3N)	get service entry by port
getservent	GETSERVENT (3N)	get service entry
herror	GETHOSTBYNAME (3N)	get network host entry error message
inet_addr	INET (3N)	Internet address manipulation routine
inet_lnaof	INET (3N)	Internet address manipulation routine
inet_makeaddr	INET (3N)	Internet address manipulation routine
inet_netof	INET (3N)	Internet address manipulation routine
inet_network	INET (3N)	Internet address manipulation routine
inet_ntoa	INET (3N)	Internet address manipulation routine
ntohl	BYTEORDER (3N)	convert host and network byte order values
ntohs	BYTEORDER (3N)	convert host and network byte order values
sethostent	GETHOSTBYNAME (3N)	set network host entry
setnetent	GETNETENT (3N)	set network entry
setprotoent	GETPROTOENT (3N)	set protocol entry
setservent	GETSERVENT (3N)	set service entry

The above synopsis applies to all of the networking library facilities. Please note this as you refer to the man pages and use the facilities in your networking routines.

NAME

htonl, htons, ntohl, ntohs - convert values between host and network byte order

SYNOPSIS

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

netlong = htonl(hostlong);
u_long netlong, hostlong;

netshort = htons(hostshort);
u_short netshort, hostshort;

hostlong = ntohl(netlong);
u_long hostlong, netlong;

hostshort = ntohs(netshort);
u_short hostshort, netshort;
```

DESCRIPTION

These routines convert 16 and 32 bit quantities between network byte order and host byte order. On machines such as ICON, these routines are defined as null macros in the include file *<netinet/in.h>*.

These routines are most often used in conjunction with Internet addresses and ports as returned by *gethostbyname*(3N) and *getservent*(3N).

SEE ALSO

gethostbyname(3N), getservent(3N)

NAME

gethostbyname, gethostbyaddr, gethostent, sethostent, endhostent, herror - get network host entry

SYNOPSIS

#include <netdb.h>

extern int h_errno;

struct hostent *gethostbyname(name)
char *name;

struct hostent *gethostbyaddr(addr, len, type)
char *addr; int len, type;

struct hostent *gethostent()

sethostent(stayopen)
int stayopen;

endhostent()

herror(string)
char *string;

DESCRIPTION

Gethostbyname and *gethostbyaddr* each return a pointer to an object with the following structure describing an internet host referenced by name or by address, respectively. This structure contains either the information obtained from the name server, *named*(1M), or broken-out fields from a line in */etc/hosts*. If the local name server is not running these routines do a lookup in */etc/hosts*.

```
struct      hostent {
      char    *h_name;           /* official name of host */
      char    **h_aliases;       /* alias list */
      int     h_addrtype;        /* host address type */
      int     h_length;          /* length of address */
      char    **h_addr_list;     /* list of addresses from name server */
};
#defineh_addr  h_addr_list[0]/* address, for backward compatibility */
```

The members of this structure are:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A zero terminated array of network addresses for the host. Host addresses are returned in network byte order.
h_addr	The first address in h_addr_list; this is for backward compatiblity.

When using the nameserver, *gethostbyname* will search for the named host in the current domain and its parents unless the name ends in a dot. If the name contains no dot, and if the environment variable ``HOSTALIASES'' contains the name of an alias file, the alias file will first be searched for an alias matching the input name. See *hostname*(5) for the domain search procedure and the alias file format.

Sethostent may be used to request the use of a connected TCP socket for queries. If the *stayopen* flag is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to *gethostbyname* or *gethostbyaddr*. Otherwise, queries are performed using UDP datagrams.

Endhostent closes the TCP connection.

DIAGNOSTICS

Error return status from *gethostbyname* and *gethostbyaddr* is indicated by return of a null pointer. The external integer *h_errno* may then be checked to see whether this is a temporary failure or an invalid or unknown host. The routine *herror* can be used to print an error message describing the failure. If its argument *string* is non-NULL, it is printed, followed by a colon and a space. The error message is printed with a trailing newline.

h_errno can have the following values:

HOST_NOT_FOUND	No such host is known.
TRY_AGAIN	This is usually a temporary error and means that the local server did not receive a response from an authoritative server. A retry at some later time may succeed.
NO_RECOVERY	Some unexpected server failure was encountered. This is a non-recoverable error.
NO_DATA	The requested name is valid but does not have an IP address; this is not a temporary error. This means that the name is known to the name server but there is no address associated with this name. Another type of request to the name server using this domain name will result in an answer; for example, a mail-forwarder may be registered for this domain.

FILES

/etc/hosts

SEE ALSO

resolver(3), hosts(4), hostname(5), named(1M)

CAVEAT

Gethostent is defined, and *sethostent* and *endhostent* are redefined, when *libc* is built to use only the routines to lookup in */etc/hosts* and not the name server.

Gethostent reads the next line of */etc/hosts*, opening the file if necessary.

Sethostent is redefined to open and rewind the file. If the *stayopen* argument is non-zero, the hosts data base will not be closed after each call to *gethostbyname* or *gethostbyaddr*. *Endhostent* is redefined to close the file.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

NAME

getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent - get network entry

SYNOPSIS

#include <netdb.h>

struct netent *getnetent()

struct netent *getnetbyname(name)
char *name;

struct netent *getnetbyaddr(net, type)
long net;
int type;

setnetent(stayopen)
int stayopen;

endnetent()

DESCRIPTION

Getnetent, *getnetbyname*, and *getnetbyaddr* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network data base, */etc/networks*.

```
struct    netent {
     char            *n_name;      /* official name of net */
     char            **n_aliases;  /* alias list */
     int             n_addrtype;   /* net number type */
     unsigned long   n_net;        /* net number */
};
```

The members of this structure are:

n_name	The official name of the network.
n_aliases	A zero terminated list of alternate names for the network.
n_addrtype	The type of the network number returned; currently only AF_INET.
n_net	The network number. Network numbers are returned in machine byte order.

Getnetent reads the next line of the file, opening the file if necessary.

Setnetent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getnetbyname* or *getnetbyaddr*.

Endnetent closes the file.

Getnetbyname and *getnetbyaddr* sequentially search from the beginning of the file until a matching net name or net address and type is found, or until EOF is encountered. Network numbers are supplied in host order.

FILES

/etc/networks

SEE ALSO

networks(4)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only Internet network numbers are currently understood. Expecting network numbers to fit in no more than 32 bits is probably naive.

NAME

getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent - get protocol entry

SYNOPSIS

#include <netdb.h>

struct protoent *getprotoent()

struct protoent *getprotobyname(name)
char *name;

struct protoent *getprotobynumber(proto)
int proto;

setprotoent(stayopen)
int stayopen

endprotoent()

DESCRIPTION

Getprotoent, *getprotobyname*, and *getprotobynumber* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network protocol data base, */etc/protocols*.

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

The members of this structure are:

p_name	The official name of the protocol.
p_aliases	A zero terminated list of alternate names for the protocol.
p_proto	The protocol number.

Getprotoent reads the next line of the file, opening the file if necessary.

Setprotoent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getprotobyname* or *getprotobynumber*.

Endprotoent closes the file.

Getprotobyname and *getprotobynumber* sequentially search from the beginning of the file until a matching protocol name or protocol number is found, or until EOF is encountered.

FILES

/etc/protocols

SEE ALSO

protocols(4)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet protocols are currently understood.

NAME

getservent, getservbyport, getservbyname, setservent, endservent - get service entry

SYNOPSIS

#include <netdb.h>

struct servent *getservent()

struct servent *getservbyname(name, proto)
char *name, *proto;

struct servent *getservbyport(port, proto)
int port; char *proto;

setservent(stayopen)
int stayopen

endservent()

DESCRIPTION

Getservent, *getservbyname*, and *getservbyport* each return a pointer to an object with the following structure containing the broken-out fields of a line in the network services data base, */etc/services*.

```
struct     servent {
     char    *s_name;         /* official name of service */
     char    **s_aliases;     /* alias list */
     int     s_port;          /* port service resides at */
     char    *s_proto;        /* protocol to use */
};
```

The members of this structure are:

s_name	The official name of the service.
s_aliases	A zero terminated list of alternate names for the service.
s_port	The port number at which the service resides. Port numbers are returned in network byte order.
s_proto	The name of the protocol to use when contacting the service.

Getservent reads the next line of the file, opening the file if necessary.

Setservent opens and rewinds the file. If the *stayopen* flag is non-zero, the net data base will not be closed after each call to *getservbyname* or .IR getservbyport .

Endservent closes the file.

Getservbyname and *getservbyport* sequentially search from the beginning of the file until a matching protocol name or port number is found, or until EOF is encountered. If a protocol name is also supplied (non-NULL), searches must also match the protocol.

FILES

/etc/services

SEE ALSO

getprotoent(3N), services(4)

DIAGNOSTICS

Null pointer (0) returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved. Expecting port numbers to fit in a 32 bit quantity is probably naive.

NAME

inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof - Internet address manipulation routines

SYNOPSIS

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

unsigned long inet_addr(cp)
char *cp;

unsigned long inet_network(cp)
char *cp;

char *inet_ntoa(in)
struct in_addr in;

struct in_addr inet_makeaddr(net, lna)
int net, lna;

int inet_lnaof(in)
struct in_addr in;

int inet_netof(in)
struct in_addr in;
```

DESCRIPTION

The routines *inet_addr* and *inet_network* each interpret character strings representing numbers expressed in the Internet standard "." notation, returning numbers suitable for use as Internet addresses and Internet network numbers, respectively. The routine *inet_ntoa* takes an Internet address and returns an ASCII string representing the address in "." notation. The routine *inet_makeaddr* takes an Internet network number and a local network address and constructs an Internet address from it. The routines *inet_netof* and *inet_lnaof* break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet address are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the "." notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right most two bytes of the network address. This makes the three part address format convenient for specifying Class B network addresses as "128.net.host".

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses as ''net.host''.

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as ''parts'' in a ''.'' notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., a leading 0x or 0X implies hexadecimal; otherwise, a leading 0 implies octal; otherwise, the number is interpreted as decimal).

SEE ALSO

gethostbyname(3N), getnetent(3N), hosts(4), networks(4),

DIAGNOSTICS

The value -1 is returned by *inet_addr* and *inet_network* for malformed requests.

BUGS

A simple way to specify Class C network addresses in a manner similar to that for Class B and Class A is needed. The string returned by *inet_ntoa* resides in a static memory area. Inet_addr should return a struct in_addr.

NAME

aliases - aliases file for sendmail

SYNOPSIS

/usr/lib/aliases

DESCRIPTION

This file describes user id aliases used by */usr/lib/sendmail.* It is formatted as a series of lines of the form

    name: name_1, name2, name_3, . . .

The *name* is the name to alias, and the *name_n* are the aliases for that name. Lines beginning with white space are continuation lines. Lines beginning with ' # ' are comments.

Aliasing occurs only on local names. Loops can not occur, since no message will be sent to any person more than once.

After aliasing has been done, local and valid recipients who have a ''.forward'' file in their home directory have messages forwarded to the list of users defined in that file.

This is only the raw data file; the actual aliasing information is placed into a binary format in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag* using the program *newaliases*(1). A *newaliases* command should be executed each time the aliases file is changed for the change to take effect.

SEE ALSO

newaliases(1), sendmail(1M)
SENDMAIL Installation and Operation Guide.
SENDMAIL An Internetwork Mail Router.

BUGS

Because of restrictions in the database routines, a single alias cannot contain more than about 1000 bytes of information. You can get longer aliases by ''chaining''; that is, make the last name in the alias be a dummy name which is a continuation alias.

NAME

hosts - host name data base

DESCRIPTION

The *hosts* file contains information regarding the known hosts on the DARPA Internet. For each host a single line should be present with the following information:

official host name
Internet address
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official host data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown hosts.

Network addresses are specified in the conventional "." notation using the *inet_addr*() routine from the Internet address manipulation library, *inet*(3N). Host names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/hosts

SEE ALSO

gethostent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

networks - network name data base

DESCRIPTION

The *networks* file contains information regarding the known networks which comprise the DARPA Internet. For each network a single line should be present with the following information:

official network name
network number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file. This file is normally created from the official network data base maintained at the Network Information Control Center (NIC), though local changes may be required to bring it up to date regarding unofficial aliases and/or unknown networks.

Network number may be specified in the conventional "." notation using the *inet_network*() routine from the Internet address manipulation library, *inet*(3N). Network names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/networks

SEE ALSO

getnetent(3N)

BUGS

A name server should be used instead of a static file.

NAME

protocols - protocol name data base

DESCRIPTION

The *protocols* file contains information regarding the known protocols used in the DARPA Internet. For each protocol a single line should be present with the following information:

official protocol name
protocol number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Protocol names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/protocols

SEE ALSO

getprotoent(3N)

BUGS

A name server should be used instead of a static file. A binary indexed file format should be available for fast access.

NAME

resolver - resolver configuration file

SYNOPSIS

/etc/resolv.conf

DESCRIPTION

The resolver configuration file contains information that is read by the resolver routines the first time they are invoked by a process. The file is designed to be human readable and contains a list of name-value pairs that provide various types of resolver information.

On a normally configured system this file should not be necessary. The only name server to be queried will be on the local machine and the domain name is retrieved from the system.

The different configuration options are:

nameserver

followed by the Internet address (in dot notation) of a name server that the resolver should query. At least one name server should be listed. Up to MAXNS (currently 3) name servers may be listed, in that case the resolver library queries tries them in the order listed. If no **nameserver** entries are present, the default is to use the name server on the local machine. (The algorithm used is to try a name server, and if the query times out, try the next, until out of name servers, then repeat trying all the name servers until a maximum number of retries are made).

domain

followed by a domain name, that is the default domain to append to names that do not have a dot in them. If no **domain** entries are present, the domain returned by *gethostname* (2) is used (everything after the first '.'). Finally, if the host name does not contain a domain part, the root domain is assumed.

The name value pair must appear on a single line, and the keyword (e.g. **nameserver**) must start the line. The value follows the keyword, separated by white space.

FILES

/etc/resolv.conf

SEE ALSO

gethostbyname(3N), resolver(3), named(1M)

NAME

services - service name data base

DESCRIPTION

The *services* file contains information regarding the known services available in the DARPA Internet. For each service a single line should be present with the following information:

official service name
port number
protocol name
aliases

Items are separated by any number of blanks and/or tab characters. The port number and protocol name are considered a single *item*; a ``/'' is used to separate the port and protocol (e.g. ``512/tcp''). A ``#'' indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines which search the file.

Service names may contain any printable character other than a field delimiter, newline, or comment character.

FILES

/etc/services

SEE ALSO

getservent(3N)

BUGS

A name server should be used instead of a static file.

NAME

hostname - host name resolution description

DESCRIPTION

Hostnames are domains, where a domain is a hierarchical, dot-separated list of subdomains; for example, the machine monet, in the Berkeley subdomain of the EDU subdomain of the ARPANET would be represented as

```
            monet.Berkeley.EDU
```

(with no trailing dot).

Hostnames are often used with network client and server programs, which must generally translate the name to an address for use. (This function is generally performed by the library routine *gethostbyname*(3).) Hostnames are resolved by the internet name resolver in the following fashion.

If the name consists of a single component, i.e. contains no dot, and if the environment variable ``HOSTALIASES'' is set to the name of a file, that file is searched for an string matching the input hostname. The file should consist of lines made up of two white-space separated strings, the first of which is the hostname alias, and the second of which is the complete hostname to be substituted for that alias. If a case-sensitive match is found between the hostname to be resolved and the first field of a line in the file, the substituted name is looked up with no further processing.

If the input name ends with a trailing dot, the trailing dot is removed, and the remaining name is looked up with no further processing.

If the input name does not end with a trailing dot, it is looked up in the local domain and its parent domains until either a match is found or fewer than 2 components of the local domain remain. For example, in the domain CS.Berkeley.EDU, the name lithium.CChem will be checked first as lithium.CChem.CS.Berkeley.EDU and then as lithium.CChem.Berkeley.EDU. Lithium.CChem.EDU will not be tried, as the there is only one component remaining from the local domain.

SEE ALSO

gethostbyname(3), resolver(4), mailaddr(5), named(1M), RFC883

NAME

mailaddr - mail addressing description

DESCRIPTION

Mail addresses are based on the ARPANET protocol listed at the end of this manual page. These addresses are in the general format

```
user@domain
```

where a domain is a hierarchical dot separated list of subdomains. For example, the address

```
eric@monet.Berkeley.ARPA
```

is normally interpreted from right to left: the message should go to the ARPA name tables (which do not correspond exactly to the physical ARPANET), then to the Berkeley gateway, after which it should go to the local host monet. When the message reaches monet it is delivered to the user ''eric''.

Unlike some other forms of addressing, this does not imply any routing. Thus, although this address is specified as an ARPA address, it might travel by an alternate route if that were more convenient or efficient. For example, at Berkeley the associated message would probably go directly to monet over the Ethernet rather than going via the Berkeley ARPANET gateway.

Abbreviation.

Under certain circumstances it may not be necessary to type the entire domain name. In general anything following the first dot may be omitted if it is the same as the domain from which you are sending the message. For example, a user on ''calder.Berkeley.ARPA'' could send to ''eric@monet'' without adding the ''.Berkeley.ARPA'' since it is the same on both sending and receiving hosts.

Certain other abbreviations may be permitted as special cases. For example, at Berkeley ARPANET hosts can be referenced without adding the ''.ARPA'' as long as their names do not conflict with a local host name.

Compatibility.

Certain old address formats are converted to the new format to provide compatibility with the previous mail system. In particular,

```
host:user
```

is converted to

```
user@host
```

to be consistent with the *rcp*(1C) command.

Also, the syntax:

```
host!user
```

is converted to:

```
user@host.UUCP
```

This is normally converted back to the ''host!user'' form before being sent on for compatibility with older UUCP hosts.

The current implementation is not able to route messages automatically through the UUCP network. Until that time you must explicitly tell the mail system which hosts to send your message through to get to your final destination.

Case Distinctions.

Domain names (i.e., anything after the "@" sign) may be given in any mixture of upper and lower case with the exception of UUCP hostnames. Most hosts accept any combination of case in user names, with the notable exception of MULTICS sites.

Differences with ARPA Protocols.

Although the UNIX addressing scheme is based on the ARPA mail addressing protocols, there are some significant differences.

At the time of this writing DARPA is converting to real domains. The following rules may be useful:

• The syntax "user@host.ARPA" is being split up into "user@host.COM", "user@host.GOV", and "user@host.EDU" for commercial, government, and educational institutions respectively.

• The syntax "user@host" (with no dots) has traditionally referred to the ARPANET. In the future this semantic will not be continued — instead, the host will be assumed to be in your organization. You should start using one of the syntaxes above.

• Host names of the form "ORG-NAME" (e.g., MIT-MC or CMU-CS-A) will be changing to "NAME.ORG.XXX" (where 'XXX' is COM, GOV, or EDU). For example, MIT-MC will change to MC.MIT.EDU. In some cases names will be split apart even if they do not have dashes. For example, USC-ISIF will probably change to F.ISI.USC.EDU.

Route-addrs.

Under some circumstances it may be necessary to route a message through several hosts to get it to the final destination. Normally this routing is done automatically, but sometimes it is desirable to route the message manually. Addresses which show these relays are termed "route-addrs." These use the syntax:

```
<@hosta,@hostb:user@hostc>
```

This specifies that the message should be sent to hosta, from there to hostb, and finally to hostc. This path is forced even if there is a more efficient path to hostc.

Route-addrs occur frequently on return addresses, since these are generally augmented by the software at each host. It is generally possible to ignore all but the "user@host" part of the address to determine the actual sender.

Postmaster.

Every site is required to have a user or user alias designated "postmaster" to which problems with the mail system may be addressed.

Other Networks.

Some other networks can be reached by giving the name of the network as the last component of the domain. *This is not a standard feature* and may not be supported at all sites. For example, messages to CSNET or BITNET sites can often be sent to "user@host.CSNET" or "user@host.BITNET" respectively.

BUGS

The RFC822 group syntax (''group:user1,user2,user3;'') is not supported except in the special case of ''group:;'' because of a conflict with old berknet-style addresses.

Route-Address syntax is grotty.

UUCP- and ARPANET-style addresses do not coexist politely.

SEE ALSO

mail(1), sendmail(1M); Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages*, RFC822.

NAME

networking - introduction to networking facilities

SYNOPSIS

#include <sys/socket.h>
#include <net/route.h>
#include <net/if.h>

DESCRIPTION

This section briefly describes the networking facilities available in the system. Documentation in this part of section 4 is broken up into two areas: *protocol families* (domains), and *protocols*. Entries describing a protocol family are marked "7N," while entries describing protocol use are marked "7P."

All network protocols are associated with a specific *protocol family*. A protocol family provides basic services to the protocol implementation to allow it to function within a specific network environment. These services may include packet fragmentation and reassembly, routing, addressing, and basic transport. A protocol family may support multiple methods of addressing, though the current protocol implementations do not. A protocol family is normally comprised of a number of protocols, one per *socket*(2) type. It is not required that a protocol family support all socket types. A protocol family may contain multiple protocols supporting the same socket abstraction.

A protocol supports one of the socket abstractions detailed in *socket*(2). A specific protocol may be accessed either by creating a socket of the appropriate type and protocol family, or by requesting the protocol explicitly when creating a socket. Protocols normally accept only one type of address format, usually determined by the addressing structure inherent in the design of the protocol family/network architecture. Certain semantics of the basic socket abstractions are protocol specific. All protocols are expected to support the basic model for their particular socket type, but may, in addition, provide non-standard facilities or extensions to a mechanism. For example, a protocol supporting the SOCK_STREAM abstraction may allow more than one byte of out-of-band data to be transmitted per out-of-band message.

A network interface is similar to a device interface. Network interfaces comprise the lowest layer of the networking subsystem, interacting with the actual transport hardware. An interface may support one or more protocol families and/or address formats. The SYNOPSIS section of each network interface entry gives a sample specification of the related drivers for use in providing a system description to the *config*(1M) program. The DIAGNOSTICS section lists messages which may appear on the console and/or in the system error log, */usr/adm/messages* (see *syslogd*(1M)), due to errors in device operation.

PROTOCOLS

The system currently supports the DARPA Internet protocols. Raw socket interfaces are provided to the IP protocol layer of the DARPA Internet, and to the IMP link layer (1822). Consult the appropriate manual pages in this section for more information regarding the support for each protocol family.

ADDRESSING

Associated with each protocol family is an address format. The following address formats are used by the system (and additional formats are defined for possible future implementation):

```
#define  AF_UNIX     1   /* local to host (pipes, portals) */
#define  AF_INET     2   /* internetwork: UDP, TCP, etc. */
#define  AF_IMPLINK  3   /* arpanet imp addresses */
#define  AF_PUP      4   /* pup protocols: e.g. BSP */
#define  AF_NS       6   /* Xerox NS protocols */
#define  AF_HYLINK   15  /* NSC Hyperchannel */
```

ROUTING

The network facilities provided limited packet routing. A simple set of data structures comprise a ''routing table'' used in selecting the appropriate network interface when transmitting packets. This table contains a single entry for each route to a specific network or host. A user process, the routing daemon, maintains this data base with the aid of two socket-specific *ioctl*(2) commands, SIOCADDRT and SIOCDELRT. The commands allow the addition and deletion of a single routing table entry, respectively. Routing table manipulations may only be carried out by super-user.

A routing table entry has the following form, as defined in *<net/route.h>*;

```
struct rtentry {
     u_long   rt_hash;
     struct   sockaddr rt_dst;
     struct   sockaddr rt_gateway;
     short    rt_flags;
     short    rt_refcnt;
     u_long   rt_use;
     struct   ifnet *rt_ifp;
};
```

with *rt_flags* defined from,

```
#define  RTF_UP       0x1  /* route usable */
#define  RTF_GATEWAY  0x2  /* destination is a gateway */
#define  RTF_HOST     0x4  /* host entry (net otherwise) */
#define  RTF_DYNAMIC  0x10 /* created dynamically (by redirect) */
```

Routing table entries come in three flavors: for a specific host, for all hosts on a specific network, for any destination not matched by entries of the first two types (a wildcard route). When the system is booted and addresses are assigned to the network interfaces, each protocol family installs a routing table entry for each interface when it is ready for traffic. Normally the protocol specifies the route through each interface as a ''direct'' connection to the destination host or network. If the route is direct, the transport layer of a protocol family usually requests the packet be sent to the same host specified in the packet. Otherwise, the interface is requested to address the packet to the gateway listed in the routing entry (i.e. the packet is forwarded).

Routing table entries installed by a user process may not specify the hash, reference count, use, or interface fields; these are filled in by the routing routines. If a route is in use when it is deleted (*rt_refcnt* is non-zero), the routing entry will be marked down and removed from the routing table, but the resources associated with it will not be reclaimed until all references to it are released. The routing code returns EEXIST if requested to duplicate an existing entry, ESRCH if requested to delete a non-existent entry, or ENOBUFS if insufficient resources were available to install a new route. User processes read the routing tables through the */dev/kmem* device. The *rt_use* field contains the number of packets sent along the route.

When routing a packet, the kernel will first attempt to find a route to the destination host. Failing that, a search is made for a route to the network of the destination. Finally, any route to a default (''wildcard'') gateway is chosen. If multiple routes are present in the table, the first route found will be used. If no entry is found, the destination is declared to be unreachable.

A wildcard routing entry is specified with a zero destination address value. Wildcard routes are used only when the system fails to find a route to the destination host and network. The combination of wildcard routes and routing redirects can provide an economical mechanism for routing traffic.

INTERFACES

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, *lo*(4), do not.

The following *ioctl* calls may be used to manipulate network interfaces. The *ioctl* is made on a socket (typically of type SOCK_DGRAM) in the desired domain. Unless specified otherwise, the request takes an *ifrequest* structure as its parameter. This structure has the form

```
struct        ifreq {
      char    ifr_name[16]; /* name of interface (e.g. "ec0") */
      union {
              struct  sockaddr ifru_addr;
              struct  sockaddr ifru_dstaddr;
              struct  sockaddr ifru_broadaddr;
              short   ifru_flags;
              int     ifru_metric;
      } ifr_ifru;
#define  ifr_addr       ifr_ifru.ifru_addr        /* address */
#define  ifr_dstaddr    ifr_ifru.ifru_dstaddr     /* other end of p-to-p link */
#define  ifr_broadaddr  ifr_ifru.ifru_broadaddr   /* broadcast address */
#define  ifr_flags      ifr_ifru.ifru_flags       /* flags */
#define  ifr_metric     ifr_ifru.ifru_metric      /* routing metric */
};
```

SIOCSIFADDR
: Set interface address for protocol family. Following the address assignment, the ''initialization'' routine for the interface is called.

SIOCGIFADDR
: Get interface address for protocol family.

SIOCSIFDSTADDR
: Set point to point address for protocol family and interface.

SIOCGIFDSTADDR
: Get point to point address for protocol family and interface.

SIOCSIFBRDADDR

Set broadcast address for protocol family and interface.

SIOCGIFBRDADDR
: Get broadcast address for protocol family and interface.

SIOCSIFFLAGS
: Set interface flags field. If the interface is marked down, any processes currently routing packets through the interface are notified; some interfaces may be reset so that incoming packets are no longer received. When marked up again, the interface is reinitialized.

SIOCGIFFLAGS
: Get interface flags.

SIOCSIFMETRIC
: Set interface routing metric. The metric is used only by user-level routers.

SIOCGIFMETRIC
: Get interface metric.

SIOCGIFCONF
: Get interface configuration list. This request takes an *ifconf* structure (see below) as a value-result parameter. The *ifc_len* field should be initially set to the size of the buffer pointed to by *ifc_buf*. On return it will contain the length, in bytes, of the configuration list.

```
/*
 * Structure used in SIOCGIFCONF request.
 * Used to retrieve interface configuration
 * for machine (useful for programs which
 * must know all networks accessible).
 */
struct    ifconf {
        int      ifc_len;  /* size of associated buffer */
        union {
                 caddr_t   ifcu_buf;
                 struct    ifreq *ifcu_req;
        } ifc_ifcu;
#define  ifc_buf ifc_ifcu.ifcu_buf    /* buffer address */
#define  ifc_req ifc_ifcu.ifcu_req    /* array of structures returned */
};
```

SEE ALSO

socket(2), ioctl(2), config(1M), routed(1M)

NAME

arp - Address Resolution Protocol

SYNOPSIS

pseudo-device ether

DESCRIPTION

ARP is a protocol used to dynamically map between DARPA Internet and 10Mb/s Ethernet addresses. It is used by all the 10Mb/s Ethernet interface drivers. It is not specific to Internet protocols or to 10Mb/s Ethernet, but this implementation currently supports only that combination.

ARP caches Internet-Ethernet address mappings. When an interface requests a mapping for an address not in the cache, ARP queues the message which requires the mapping and broadcasts a message on the associated network requesting the address mapping. If a response is provided, the new mapping is cached and any pending message is transmitted. ARP will queue at most one packet while waiting for a mapping request to be responded to; only the most recently ''transmitted'' packet is kept.

To facilitate communications with systems which do not use ARP, *ioctl*s are provided to enter and delete entries in the Internet-to-Ethernet tables. Usage:

```
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
struct arpreq arpreq;

ioctl(s, SIOCSARP, (caddr_t)&arpreq);
ioctl(s, SIOCGARP, (caddr_t)&arpreq);
ioctl(s, SIOCDARP, (caddr_t)&arpreq);
```

Each ioctl takes the same structure as an argument. SIOCSARP sets an ARP entry, SIOCGARP gets an ARP entry, and SIOCDARP deletes an ARP entry. These ioctls may be applied to any socket descriptor *s*, but only by the super-user. The *arpreq* structure contains:

```
/*
 * ARP ioctl request
 */
struct arpreq {
        struct sockaddr   arp_pa;      /* protocol address */
        struct sockaddr   arp_ha;      /* hardware address */
        int               arp_flags;   /* flags */
};
/*  arp_flags field values */
#define  ATF_COM           0x02  /* completed entry (arp_ha.valid) */
#define  ATF_PERM          0x04  /* permanent entry */
#define  ATF_PUBL          0x08  /* publish (respond for other host) */
#define  ATF_USETRAILERS   0x10  /* send trailer packets to host */
```

The address family for the *arp_pa* sockaddr must be AF_INET; for the *arp_ha* sockaddr it must be AF_UNSPEC. The only flag bits which may be written are ATF_PERM, ATF_PUBL and ATF_USETRAILERS. ATF_PERM causes the entry to be permanent if the ioctl call succeeds. The peculiar nature of the ARP tables may cause the ioctl to fail if more than 8 (permanent) Internet host addresses hash to the same slot. ATF_PUBL specifies that the ARP code should respond to ARP requests for the indicated host coming from other machines. This allows a host to act as an "ARP server," which may be useful in convincing an ARP-only machine to talk to a non-ARP machine.

ARP is also used to negotiate the use of trailer IP encapsulations; trailers are an alternate encapsulation used to allow efficient packet alignment for large packets despite variable-sized headers. Hosts which wish to receive trailer encapsulations so indicate by sending gratuitous ARP translation replies along with replies to IP requests; they are also sent in reply to IP translation replies. The negotiation is thus fully symmetrical, in that either or both hosts may request trailers. The ATF_USETRAILERS flag is used to record the receipt of such a reply, and enables the transmission of trailer packets to that host.

ARP watches passively for hosts impersonating the local host (i.e. a host which responds to an ARP mapping request for the local host's address).

DIAGNOSTICS

duplicate IP address!! sent from ethernet address: %x:%x:%x:%x:%x:%x. ARP has discovered another host on the local network which responds to mapping requests for its own Internet address.

SEE ALSO

inet(7P), arp(1M), ifconfig(1M)
"An Ethernet Address Resolution Protocol," RFC826, Dave Plummer, Network Information Center, SRI.
"Trailer Encapsulations," RFC893, S.J. Leffler and M.J. Karels, Network Information Center, SRI.

BUGS

ARP packets on the Ethernet use only 42 bytes of data; however, the smallest legal Ethernet packet is 60 bytes (not including CRC). Some systems may not enforce the minimum packet size, others will.

NAME

icmp - Internet Control Message Protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, proto);

DESCRIPTION

ICMP is the error and control message protocol used by IP and the Internet protocol family. It may be accessed through a ''raw socket'' for network monitoring and diagnostic functions. The *proto* parameter to the socket call to create an ICMP socket is obtained from *getprotobyname*(3N). ICMP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the *connect*(2) call may also be used to fix the destination for future packets (in which case the *read*(2) or *recv*(2) and *write*(2) or *send*(2) system calls may be used).

Outgoing packets automatically have an IP header prepended to them (based on the destination address). Incoming packets are received with the IP header and options intact.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
[ENOTCONN]	when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

SEE ALSO

send(2), recv(2), intro(7N), inet(7P), ip(7P)

NAME

ip - Internet Protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_RAW, proto);

DESCRIPTION

IP is the transport layer protocol used by the Internet protocol family. Options may be set at the IP level when using higher-level protocols that are based on IP (such as TCP and UDP). It may also be accessed through a ''raw socket'' when developing new protocols, or special purpose applications.

A single generic option is supported at the IP level, IP_OPTIONS, that may be used to provide IP options to be transmitted in the IP header of each outgoing packet. Options are set with *setsockopt*(2) and examined with *getsockopt*(2). The format of IP options to be sent is that specified by the IP protocol specification, with one exception: the list of addresses for Source Route options must include the first-hop gateway at the beginning of the list of gateways. The first-hop gateway address will be extracted from the option list and the size adjusted accordingly before use. IP options may be used with any socket type in the Internet family.

Raw IP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the *connect*(2) call may also be used to fix the destination for future packets (in which case the *read*(2) or *recv*(2) and *write*(2) or *send*(2) system calls may be used).

If *proto* is 0, the default protocol IPPROTO_RAW is used for outgoing packets, and only incoming packets destined for that protocol are received. If *proto* is non-zero, that protocol number will be used on outgoing packets and to filter incoming packets.

Outgoing packets automatically have an IP header prepended to them (based on the destination address and the protocol number the socket is created with). Incoming packets are received with IP header and options intact.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
[ENOTCONN]	when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

The following errors specific to IP may occur when setting or getting IP options:

[EINVAL]	An unknown socket option name was given.

[EINVAL]	The IP option field was improperly formed; an option field was shorter than the minimum value or longer than the option buffer provided.

SEE ALSO

getsockopt(2), send(2), recv(2), intro(7N), icmp(7P), inet(7P)

NAME

tcp - Internet Transmission Control Protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_STREAM, 0);

DESCRIPTION

The TCP protocol provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP uses the standard Internet address format and, in addition, provides a per-host collection of "port addresses". Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets utilizing the tcp protocol are either "active" or "passive". Active sockets initiate connections to passive sockets. By default TCP sockets are created active; to create a passive socket the *listen*(2) system call must be used after binding the socket with the *bind*(2) system call. Only passive sockets may use the *accept*(2) call to accept incoming connections. Only active sockets may use the *connect*(2) call to initiate connections.

Passive sockets may "underspecify" their location to match incoming connection requests from multiple networks. This technique, termed "wildcard addressing", allows a single server to provide service to clients on multiple networks. To create a socket which listens on all networks, the Internet address INADDR_ANY must be bound. The TCP port may still be specified at this time; if the port is not specified the system will assign one. Once a connection has been established the socket's address is fixed by the peer entity's location. The address assigned the socket is the address associated with the network interface through which packets are being transmitted and received. Normally this address corresponds to the peer entity's network.

TCP supports one socket option which is set with *setsockopt*(2) and tested with *getsockopt*(2). Under most circumstances, TCP sends data when it is presented; when outstanding data has not yet been acknowledged, it gathers small amounts of output to be sent in a single packet once an acknowledgement is received. For a small number of clients, such as window systems that send a stream of mouse events which receive no replies, this packetization may cause significant delays. Therefore, TCP provides a boolean option, TCP_NODELAY (from *<netinet/tcp.h>*, to defeat this algorithm. The option level for the *setsockopt* call is the protocol number for TCP, available from *getprotobyname*(3N).

Options at the IP transport level may be used with TCP; see *ip*(4P). Incoming connection requests that are source-routed are noted, and the reverse source route is used in responding.

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[ETIMEDOUT]	when a connection was dropped due to excessive retransmissions;
[ECONNRESET]	when the remote peer forces the connection to be closed;

[ECONNREFUSED]	when the remote peer actively refuses connection establishment (usually because no process is listening to the port);
[EADDRINUSE]	when an attempt is made to create a socket with a port which has already been allocated;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

SEE ALSO

getsockopt(2), socket(2), intro(7N), inet(7P), ip(7P)

NAME

udp - Internet User Datagram Protocol

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>

s = socket(AF_INET, SOCK_DGRAM, 0);

DESCRIPTION

UDP is a simple, unreliable datagram protocol which is used to support the SOCK_DGRAM abstraction for the Internet protocol family. UDP sockets are connectionless, and are normally used with the *sendto* and *recvfrom* calls, though the *connect*(2) call may also be used to fix the destination for future packets (in which case the *recv*(2) or *read*(2) and *send*(2) or *write(2)* system calls may be used).

UDP address formats are identical to those used by TCP. In particular UDP provides a port identifier in addition to the normal Internet address format. Note that the UDP port space is separate from the TCP port space (i.e. a UDP port may not be ''connected'' to a TCP port). In addition broadcast packets may be sent (assuming the underlying network supports this) by using a reserved ''broadcast address''; this address is network interface dependent.

Options at the IP transport level may be used with UDP; see *ip*(4P).

DIAGNOSTICS

A socket operation may fail with one of the following errors returned:

[EISCONN]	when trying to establish a connection on a socket which already has one, or when trying to send a datagram with the destination address specified and the socket is already connected;
[ENOTCONN]	when trying to send a datagram, but no destination address is specified, and the socket hasn't been connected;
[ENOBUFS]	when the system runs out of memory for an internal data structure;
[EADDRINUSE]	when an attempt is made to create a socket with a port which has already been allocated;
[EADDRNOTAVAIL]	when an attempt is made to create a socket with a network address for which no network interface exists.

SEE ALSO

getsockopt(2), recv(2), send(2), socket(2), intro(7N), inet(7P), ip(7P)

Appendix B – SENDMAIL – An Internetwork Mail Router

The following appendix contains a document by Eric Allman titled "SENDMAIL – An Internetwork Mail Router". This document provides and overview of the SENDMAIL program as well as the functional theory behind the program as it was originally developed for use at the University of California at Berkeley.

This document is taken in whole from the System Manager's Manual for the 4.3BSD version of the UNIX® operating system, which is a subset of the ICON/UXV implementation of AT&T's UNIX System V operating system.

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 organization of the address space.

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

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

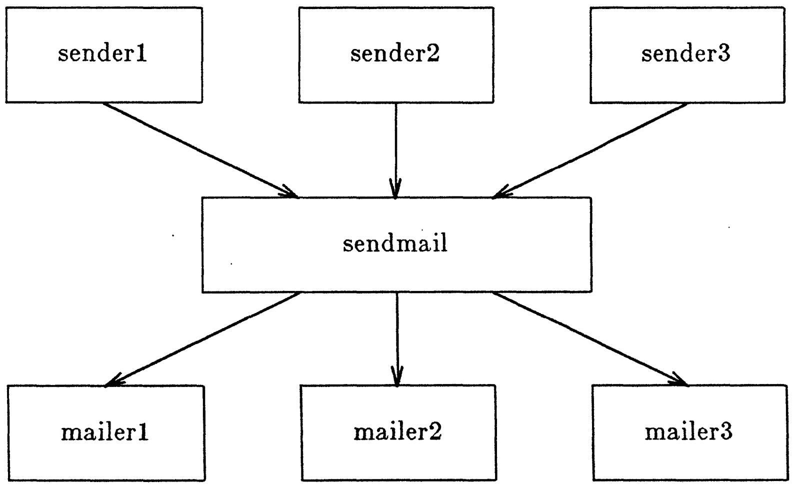

Figure 1 – Sendmail System Structure.

[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:

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

[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.

(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"

will use a different incoming mailer.

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

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 if 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[4].

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

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.

---

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

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

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 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

[6]This is similar to the NBS standard.

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 C – SENDMAIL Installation and Operating Guide

The following appendix contains a document by Eric Allman, from the University of California at Berkeley, title "*SENDMAIL Installation and Operating Guide*". This document provides information on how to install a basic version of *sendmail*, the Internetwork mail routing program. This document is a logical extension of the document "SENDMAIL – An Internetwork Mail Router" found in Appendix B.

SENDMAIL

INSTALLATION AND OPERATION GUIDE

Eric Allman
Britton-Lee, Inc.

Version 5.8

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.

*UNIX is a trademark of Bell Laboratories.

TABLE OF CONTENTS

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, normally */usr/src/usr.lib/sendmail* on 4.3BSD.

1.1. Off-The-Shelf Configurations

The configuration files are all in the subdirectories *cf.named* and *cf.hosttable* of the sendmail directory. The directory *cf.named* contains configuration files that have been tailored for the name server *named*(8). These are the configuration files currently being used at Berkeley. The configuration files in *cf.hosttable* are some typical ones and the old Berkeley versions from before the name server was being used. You should create a symbolic link from *cf* to the directory that you are going to use. For example, to use the name server:

```
ln –s cf.named cf
```

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.

Three off the shelf configurations are supplied to handle the basic cases:

(1) Arpanet (TCP) sites not running the name server can use *cf.hosttable/arpaproto.cf.* For simple sites, you should be able to use this file without modification. This file is not in *m4* format.

(2) UUCP sites can use *cf.hosttable/uucpproto.cf.* If your UUCP node name is not the same as your system name (as printed by the *hostname*(1) command) you may have to modify the **U** macro. This file is not in *m4* format.

(3) A group of machines at a single site connected by an ethernet with (only) one host connected to the outside world via UUCP is represented by two configuration files: *cf.hosttable/lanroot.mc* should be installed on the host with outside connections and *cf.hosttable/lanleaf.mc* should be installed on all other hosts. These will require slightly more configuration. First, in both files the **D** macro and **D** class must be adjusted to indicate your local domain. For example, if your company is known as "Muse" you will want to change both of those accordingly. (As distributed, they are called XXX.) Second, in *lanleaf.mc* you will have to change the **R** macro to the name of the root host, that is, the host that runs *lanroot.mc*. For example, they might appear as:

```
DDMuse
CDLOCAL Muse

DRErato
```

Internally, the root host will be known as "Erato.Muse" and other hosts will be known as "Thalia.Muse", "Clio.Muse", etc.

The file you need should be copied to a file with the same name as your system, e.g.,

```
cp uucpproto.cf ucsfcgl.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 4.3BSD system. It may have to be slightly tailored for use on other systems.

Before using this makefile, you should create a symbolic link from *cf* to the directory containing your configuration files. You should also 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
make installcf
```

in the root directory of the *sendmail* distribution. This will make all binaries and install them in the standard places. The second and third *make* commands 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 4.3BSD directory code and do not have the compatibility routines installed in your system library, you should execute the command:

```
(cd lib; make ndir)
```

This will compile and install the 4.3 compatibility routines in the library. You should then type:

```
(cd lib; 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
make clean
make
cp sendmail /usr/lib
chgrp kmem /usr/lib/sendmail
```

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. /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.

1.3.12. /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 *syslogd*(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 *syslogd*(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 4.6.

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.

E An error address. If any such lines exist, they represent the addresses that should receive error messages.

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 changes 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)
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.EDU
```

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 maintenance 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:

```
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.

[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.

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, that is, when successfully delivered to a mailer with 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).

Most of these options default appropriately for most sites. However, sites having very high mail loads may find they need to tune them as appropriate for their mail load. In particular, sites experiencing a large number of small messages, many of which are delivered to many recipients, may find that they need to adjust the parameters dealing with queue priorities.

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 fifteen minutes and one 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. Forking During Queue Runs

By setting the **Y** option, *sendmail* will fork before each individual message while running the queue. This will prevent *sendmail* from consuming large amounts of memory, so it may be useful in memory-poor environments. However, if the **Y** option is not set, *sendmail* will keep track of hosts that are down during a queue run, which can improve performance dramatically.

4.3. Queue Priorities

Every message is assigned a priority when it is first instantiated, consisting of the message size (in bytes) offset by the message class times the "work class factor" and the number of recipients times the "work recipient factor." The priority plus the creation time of the message (in seconds since January 1, 1970) are used to order the queue. Higher numbers for the priority mean that the message will be processed later when running the queue.

The message size is included so that large messages are penalized relative to small messages. The message class allows users to send "high priority" messages by including a "Precedence:" field in their message; the value of this field is looked up in the **P** lines of the configuration file. Since the number of recipients affects the amount of load a message presents to the system, this is also included into the priority.

The recipient and class factors can be set in the configuration file using the **y** and **z** options respectively. They default to 1000 (for the recipient factor) and 1800 (for the class factor). The initial priority is:

$$pri = size - (class * z) + (nrcpt * y)$$

(Remember, higher values for this parameter actually mean that the job will be treated with lower priority.)

The priority of a job can also be adjusted each time it is processed (that is, each time an attempt is made to deliver it) using the "work time factor," set by the **Z** option. This is added to the priority, so it normally decreases the precedence of the job, on the grounds that jobs that have failed many times will tend to fail again in the future.

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) if the *Queue Factor* (**q** option) divided by the difference in the current load average and the **x** option plus one exceeds the priority of the message — that is, the message is queued iff:

$$pri > \frac{QF}{LA - x + 1}$$

The **q** option defaults to 10000, so each point of load average is worth 10000 priority points (as described above, that is, bytes + seconds + offsets).

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

4.5. 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.

4.6. 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 deferred (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.

4.7. 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.

4.7.1. 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.

4.7.2. 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).

4.7.3. 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*

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*.

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:

P*name=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.

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.

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 "@" were in the **$o** macro, then the input "a@b" would be scanned as three tokens: "a," "@," and "b." Finally, the **$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.

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 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 using the *gethostent*(3) routines and replaced by the canonical name. For example, "$[csam$]" would become "lbl-csam.arpa" and "$[[128.32.130.2]$]" would become "vangogh.berkeley.edu."

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 preceded 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 precede 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:

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.

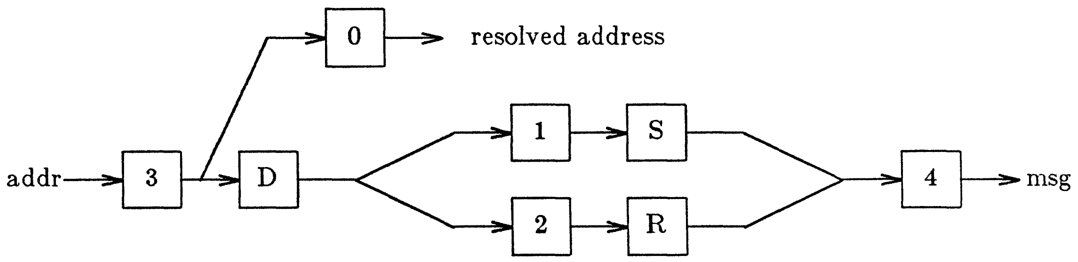

Figure 2 – Rewriting set semantics
D – sender domain addition
S – mailer-specific sender rewriting
R – mailer-specific recipient rewriting

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 and multiple mail connections. 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." When monet receives mail for delivery, it checks whether it knows that the destination host is directly reachable; if so, mail is sent to that host. 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 mail connection is added, the only host that *must* have its tables updated is ucbvax; the others *may* be updated if convenient, but this is not critical.

This picture is slightly muddied due to network connections that are not actually located on ucbvax. For example, some UUCP connections are 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 UUCP host is transferred via the ethernet from monet to ucbvax, then via the ethernet from ucbvax to ucbarpa, and then is submitted to UUCP. 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 appropriate UUCP mail directly to ucbarpa if the load got too high; if monet failed to note a host as connected to ucbarpa 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, for example, if ucbarpa thought that ucbvax had the UUCP 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 and few external connections) 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 might be "a.CC.Berkeley.EDU"; reading from right to left, "EDU" is a top level domain comprising educational institutions, "Berkeley" is a logical domain name, "CC" represents the Computer Center, (in this case a strictly logical entity), and "a" is a host in the Computer Center.

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.cf" 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 $g" in the argv template. If the mailer must be called as **root** the "S" flag should be given; this will not reset the userid before calling the mailer[3]. If this mailer is local (i.e., will perform final delivery rather than another network hop) the "l" flag should be given. Quote characters (backslashes and " marks) can be stripped from addresses if the "s" flag is specified; if this is not given they are passed through. If the mailer is capable of sending to more than one user on the same host in a single transaction the "m" flag should be stated. If this flag is on, then the argv template containing **$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[4].

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."

[3]*Sendmail* must be running setuid to root for this to work.

[4]The "c" configuration option must be given for this to be effective.

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 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.
−F*name*	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.
−b*x*	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>.

−q*time*	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.
−C*file*	Use a different configuration file. *Sendmail* runs as the invoking user (rather than root) when this flag is specified.
−d*level*	Set debugging level.
−o*x 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.

A*file*	Use the named *file* as the alias file. If no file is specified, use *aliases* in the current directory.
a*N*	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.
B*c*	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.
d*x*	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**.
e*x*	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
F*n*	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.
g*n*	Set the default group id for mailers to run in to *n*.
H*file*	Specify the help file for SMTP.
i	Ignore dots in incoming messages.
L*n*	Set the default log level to *n*.
M*x 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.
N*netname*	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.
Q*dir*	Use the named *dir* as the queue directory.
q*factor*	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.
r*time*	Timeout reads after *time* interval.
S*file*	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.
T*time*	Set the queue timeout to *time*. After this interval, messages that have not been successfully sent will be returned to the sender.
t*S,D*	Set the local time zone name to *S* for standard time and *D* for daylight time; this is only used under version six.
u*n*	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.
x*LA*	When the system load average exceeds *LA*, just queue messages (i.e., don't try to send them).
X*LA*	When the system load average exceeds *LA*, refuse incoming SMTP connections.
y*fact*	The indicated *fact*or is added to the priority (thus *lowering* the priority of the job) for each recipient, i.e., this value penalizes jobs with large numbers of recipients.
Y	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.
z*fact*	The indicated *fact*or is multiplied by the message class (determined by the Precedence: field in the user header and the **P** lines in the configuration file) and subtracted from the priority. Thus, messages with a higher Priority: will be favored.
Z*fact*	The *fact*or is added to the priority every time a job is processed. Thus, each time a job is processed, its priority will be decreased by the indicated value. In most environments this should be positive, since hosts that are down are all too often down for a long time.

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 if 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 [1024]	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 [256]	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 [17]	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).

MAXUSERENVIRON [40]
The maximum number of items in the user environment that will be passed to subordinate mailers.

QUEUESIZE [600] The maximum number of entries that will be processed in a single queue run.

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.

NDBM If set, the new version of the DBM library that allows multiple databases will be used. "DBM" must also be set.

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 4.2 or 4.3BSD.

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 (permanent) 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, that is, errors that are likely to go away without explicit action on the part of a systems administrator. 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 suppress 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.

Configuration in src/daemon.c

The file *src/daemon.c* contains a number of routines that are dependent on the local networking environment. The version supplied is specific to 4.3 BSD.

The routine *maphostname* is called to convert strings within **$**[... **$**] symbols. It can be modified if you wish to provide a more sophisticated service, e.g., mapping UUCP host names to full paths.

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.
/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.

Appendix D – Introduction to the Internet Protocols

The following appendix contains a document from RUTGERS, the State University of New Jersey, titled "*Introduction to the Internet Protocols*". The document provides an introduction to TCP/IP, giving a reasonable explanation of the capabilities of the Internet Protocols. References are made to "RFC" and "IEN" documents throughout this document. Information is provided to allow you to request copies of the referenced documents

Introduction to the Internet Protocols

C R

C S
Computer Science Facilities Group
C I

L S

RUTGERS
The State University of New Jersey
Center for Computers and Information Services
Laboratory for Computer Science Research

This is an introduction to the Internet networking protocols (TCP/IP). It includes a summary of the facilities available and brief descriptions of the major protocols in the family.

† Unix is a trademark of AT&T Technologies, Inc.

This document is a brief introduction to TCP/IP, followed by advice on what to read for more information. This is not intended to be a complete description. It can give you a reasonable idea of the capabilities of the protocols. But if you need to know any details of the technology, you will want to read the standards yourself. Throughout the text, you will find references to the standards, in the form of "RFC" or "IEN" numbers. These are document numbers. The final section of this document tells you how to get copies of those standards.

TABLE OF CONTENTS

Introduction to Internet Protocols

1. What is TCP/IP?

TCP/IP is a set of protocols developed to allow cooperating computers to share resources across a network. It was developed by a community of researchers centered around the ARPAnet. Certainly the ARPAnet is the best-known TCP/IP network. However as of June, 87, at least 130 different vendors had products that support TCP/IP, and thousands of networks of all kinds use it.

First some basic definitions. The most accurate name for the set of protocols we are describing is the "Internet protocol suite". TCP and IP are two of the protocols in this suite. (They will be described below.) Because TCP and IP are the best known of the protocols, it has become common to use the term TCP/IP or IP/TCP to refer to the whole family. It is probably not worth fighting this habit. However this can lead to some oddities. For example, I find myself talking about NFS as being based on TCP/IP, even though it doesn't use TCP at all. (It does use IP. But it uses an alternative protocol, UDP, instead of TCP. All of this alphabet soup will be unscrambled in the following pages.)

The Internet is a collection of networks, including the Arpanet, NSFnet, regional networks such as NYsernet, local networks at a number of University and research institutions, and a number of military networks. The term "Internet" applies to this entire set of networks. The subset of them that is managed by the Department of Defense is referred to as the "DDN" (Defense Data Network). This includes some research-oriented networks, such as the Arpanet, as well as more strictly military ones. (Because much of the funding for Internet protocol developments is done via the DDN organization, the terms Internet and DDN can sometimes seem equivalent.) All of these networks are connected to each other. Users can send messages from any of them to any other, except where there are security or other policy restrictions on access. Officially speaking, the Internet protocol documents are simply standards adopted by the Internet community for its own use. More recently, the Department of Defense issued a MILSPEC definition of TCP/IP. This was intended to be a more formal definition, appropriate for use in purchasing specifications. However most of the TCP/IP community continues to use the Internet standards. The MILSPEC version is intended to be consistent with it.

Whatever it is called, TCP/IP is a family of protocols. A few provide "low-level" functions needed for many applications. These include IP, TCP, and UDP. (These will be described in a bit more detail later.) Others are protocols for doing specific tasks, e.g. transferring files between computers, sending mail, or finding out who is logged in on another computer. Initially TCP/IP was used mostly between minicomputers or mainframes. These machines had their own disks, and generally were self-contained. Thus the most important "traditional" TCP/IP services are:

- *file transfer*. The file transfer protocol (FTP) allows a user on any computer to get files from another computer, or to send files to another computer. Security is handled by requiring the user to specify a user name and password for the other computer. Provisions are made for handling file transfer between machines with different character set, end of line conventions, etc. This is not quite the same thing as more recent "network file system" or "netbios" protocols, which will be described below. Rather, FTP is a utility that you run any time you want to access a file on another system. You use it to copy the file to your own system. You then work with the local copy. (See RFC 959 for specifications for FTP.)

- *remote login.* The network terminal protocol (TELNET) allows a user to log in on any other computer on the network. You start a remote session by specifying a computer to connect to. From that time until you finish the session, anything you type is sent to the other computer. Note that you are really still talking to your own computer. But the telnet program effectively makes your computer invisible while it is running. Every character you type is sent directly to the other system. Generally, the connection to the remote computer behaves much like a dialup connection. That is, the remote system will ask you to log in and give a password, in whatever manner it would normally ask a user who had just dialed it up. When you log off of the other computer, the telnet program exits, and you will find yourself talking to your own computer. Microcomputer implementations of telnet generally include a terminal emulator for some common type of terminal. (See RFC's 854 and 855 for specifications for telnet. By the way, the telnet protocol should not be confused with Telenet, a vendor of commercial network services.)
- *computer mail.* This allows you to send messages to users on other computers. Originally, people tended to use only one or two specific computers. They would maintain "mail files" on those machines. The computer mail system is simply a way for you to add a message to another user's mail file. There are some problems with this in an environment where microcomputers are used. The most serious is that a micro is not well suited to receive computer mail. When you send mail, the mail software expects to be able to open a connection to the addressee's computer, in order to send the mail. If this is a microcomputer, it may be turned off, or it may be running an application other than the mail system. For this reason, mail is normally handled by a larger system, where it is practical to have a mail server running all the time. Microcomputer mail software then becomes a user interface that retrieves mail from the mail server. (See RFC 821 and 822 for specifications for computer mail. See RFC 937 for a protocol designed for microcomputers to use in reading mail from a mail server.)

These services should be present in any implementation of TCP/IP, except that micro-oriented implementations may not support computer mail. These traditional applications still play a very important role in TCP/IP-based networks. However more recently, the way in which networks are used has been changing. The older model of a number of large, self-sufficient computers is beginning to change. Now many installations have several kinds of computers, including microcomputers, workstations, minicomputers, and mainframes. These computers are likely to be configured to perform specialized tasks. Although people are still likely to work with one specific computer, that computer will call on other systems on the net for specialized services. This has led to the "server/client" model of network services. A server is a system that provides a specific service for the rest of the network. A client is another system that uses that service. (Note that the server and client need not be on different computers. They could be different programs running on the same computer.) Here are the kinds of servers typically present in a modern computer setup. Note that these computer services can all be provided within the framework of TCP/IP.

- *network file systems.* This allows a system to access files on another computer in a somewhat more closely integrated fashion than FTP. A network file system provides the illusion that disks or other devices from one system are directly connected to other systems. There is no need to use a special network utility to access a file on another system. Your computer simply thinks it has some extra disk drives. These extra "virtual" drives refer to the other system's disks. This capability is useful for several different purposes. It lets you put large disks on a few computers, but still give others access to the disk space. Aside from the obvious economic benefits, this allows people working on several computers to share common files. It makes system maintenance and backup easier, because you don't have to worry about updating and backing up copies on lots of different machines. A number of vendors now offer high-performance diskless computers. These computers have no disk drives at all. They are entirely dependent upon disks attached to common "file servers". (See RFC's 1001 and 1002 for a description of PC-oriented NetBIOS over TCP. In the workstation and minicomputer area, Sun's Network File System is more likely to be used. Protocol specifications for it are available from Sun Microsystems.)

- *remote printing.* This allows you to access printers on other computers as if they were directly attached to yours. (The most commonly used protocol is the remote lineprinter protocol from Berkeley Unix. Unfortunately, there is no protocol document for this. However the C code is easily obtained from Berkeley, so implementations are common.)
- *remote execution.* This allows you to request that a particular program be run on a different computer. This is useful when you can do most of your work on a small computer, but a few tasks require the resources of a larger system. There are a number of different kinds of remote execution. Some operate on a command by command basis. That is, you request that a specific command or set of commands should run on some specific computer. (More sophisticated versions will choose a system that happens to be free.) However there are also "remote procedure call" systems that allow a program to call a subroutine that will run on another computer. (There are many protocols of this sort. Berkeley Unix contains two servers to execute commands remotely: rsh and rexec. The man pages describe the protocols that they use. The user-contributed software with Berkeley 4.3 contains a "distributed shell" that will distribute tasks among a set of systems, depending upon load. Remote procedure call mechanisms have been a topic for research for a number of years, so many organizations have implementations of such facilities. The most widespread commercially-supported remote procedure call protocols seem to be Xerox's Courier and Sun's RPC. Protocol documents are available from Xerox and Sun. There is a public implementation of Courier over TCP as part of the user-contributed software with Berkeley 4.3. An implementation of RPC was posted to Usenet by Sun, and also appears as part of the user-contributed software with Berkeley 4.3.)
- *name servers.* In large installations, there are a number of different collections of names that have to be managed. This includes users and their passwords, names and network addresses for computers, and accounts. It becomes very tedious to keep this data up to date on all of the computers. Thus the databases are kept on a small number of systems. Other systems access the data over the network. (RFC 822 and 823 describe the name server protocol used to keep track of host names and Internet addresses on the Internet. This is now a required part of any TCP/IP implementation. IEN 116 describes an older name server protocol that is used by a few terminal servers and other products to look up host names. Sun's Yellow Pages system is designed as a general mechanism to handle user names, file sharing groups, and other databases commonly used by Unix systems. It is widely available commercially. Its protocol definition is available from Sun.)
- *terminal servers.* Many installations no longer connect terminals directly to computers. Instead they connect them to terminal servers. A terminal server is simply a small computer that only knows how to run telnet (or some other protocol to do remote login). If

your terminal is connected to one of these, you simply type the name of a computer, and you are connected to it. Generally it is possible to have active connections to more than one computer at the same time. The terminal server will have provisions to switch between connections rapidly, and to notify you when output is waiting for another connection. (Terminal servers use the telnet protocol, already mentioned. However any real terminal server will also have to support name service and a number of other protocols.)

- *network-oriented window systems.* Until recently, high-performance graphics programs had to execute on a computer that had a bit-mapped graphics screen directly attached to it. Network window systems allow a program to use a display on a different computer. Full-scale network window systems provide an interface that lets you distribute jobs to the systems that are best suited to handle them, but still give you a single graphically-based user interface. (The most widely-implemented window system is X. A protocol description is available from MIT's Project Athena. A reference implementation is publically available from MIT. A number of vendors are also supporting NeWS, a window system defined by Sun. Both of these systems are designed to use TCP/IP.)

Note that some of the protocols described above were designed by Berkeley, Sun, or other organizations. Thus they are not officially part of the Internet protocol suite. However they are implemented using TCP/IP, just as normal TCP/IP application protocols are. Since the protocol definitions are not considered proprietary, and since commercially-support implementations are widely available, it is reasonable to think of these protocols as being effectively part of the Internet suite. Note that the list above is simply a sample of the sort of services available through TCP/IP. However it does contain the majority of the "major" applications. The other commonly-used protocols tend to be specialized facilities for getting information of various kinds, such as who is logged in, the time of day, etc. However if you need a facility that is not listed here, we encourage you to look through the current edition of Internet Protocols (currently RFC 1011), which lists all of the available protocols, and also to look at some of the major TCP/IP implementations to see what various vendors have added.

2. General description of the TCP/IP protocols

TCP/IP is a layered set of protocols. In order to understand what this means, it is useful to look at an example. A typical situation is sending mail. First, there is a protocol for mail. This defines a set of commands which one machine sends to another, e.g. commands to specify who the sender of the message is, who it is being sent to, and then the text of the message. However this protocol assumes that there is a way to communicate reliably between the two computers. Mail, like other application protocols, simply defines a set of commands and messages to be sent. It is designed to be used together with TCP and IP. TCP is responsible for making sure that the commands get through to the other end. It keeps track of what is sent, and retransmitts anything that did not get through. If any message is too large for one datagram, e.g. the text of the mail, TCP will split it up into several datagrams, and make sure that they all arrive correctly. Since these functions are needed for many applications, they are put together into a separate protocol, rather than being part of the specifications for sending mail. You can think of TCP as forming a library of routines that applications can use when they need reliable network communications with another computer. Similarly, TCP calls on the services of IP. Although the services that TCP supplies are needed by many applications, there are still some kinds of applications that don't need them. However there are some services that every application needs. So these services are put together into IP. As with TCP, you can think of IP as a library of routines that TCP calls on, but which is also available to applications that don't use TCP. This strategy of building several levels of protocol is called "layering". We think of the applications programs such as mail, TCP, and IP, as being separate "layers", each of which calls on the services of the layer below it. Generally, TCP/IP applications use 4 layers:

- an application protocol such as mail
- a protocol such as TCP that provides services need by many applications
- IP, which provides the basic service of getting datagrams to their destination
- the protocols needed to manage a specific physical medium, such as Ethernet or a point to point line.

TCP/IP is based on the "catenet model". (This is described in more detail in IEN 48.) This model assumes that there are a large number of independent networks connected together by gateways. The user should be able to access computers or other resources on any of these networks. Datagrams will often pass through a dozen different networks before getting to their final destination. The routing needed to accomplish this should be completely invisible to the user. As far as the user is concerned, all he needs to know in order to access another system is an "Internet address". This is an address that looks like 128.6.4.194. It is actually a 32-bit number. However it is normally written as 4 decimal numbers, each representing 8 bits of the address. (The term "octet" is used by Internet documentation for such 8-bit chunks. The term "byte" is not used, because TCP/IP is supported by some computers that have byte sizes other than 8 bits.) Generally the structure of the address gives you some information about how to get to the system. For example, 128.6 is a network number assigned by a central authority to Rutgers University. Rutgers uses the next octet to indicate which of the campus Ethernets is involved. 128.6.4 happens to be an Ethernet used by the Computer Science Department. The last octet allows for up to 254 systems on each Ethernet. (It is 254 because 0 and 255 are not allowed, for reasons that will be discussed later.) Note that 128.6.4.194 and 128.6.5.194 would be different systems. The structure of an Internet address is described in a bit more detail later.

Of course we normally refer to systems by name, rather than by Internet address. When we specify a name, the network software looks it up in a database, and comes up with the corresponding Internet address. Most of the network software deals strictly in terms of the address. (RFC 882 describes the name server technology used to handle this lookup.)

TCP/IP is built on "connectionless" technology. Information is transfered as a sequence of "datagrams". A datagram is a collection of data that is sent as a single message. Each of these datagrams is sent through the network individually. There are provisions to open connections (i.e. to start a conversation that will continue for some time). However at some level, information from those connections is broken up into datagrams, and those datagrams are treated by the network as completely separate. For example, suppose you want to transfer a 15000 octet file. Most networks can't handle a 15000 octet datagram. So the protocols will break this up into something like 30 500-octet datagrams. Each of these datagrams will be sent to the other end. At that point, they will be put back together into the 15000-octet file. However while those datagrams are in transit, the network doesn't know that there is any connection between them. It is perfectly possible that datagram 14 will actually arrive before datagram 13. It is also possible that somewhere in the network, an error will occur, and some datagram won't get through at all. In that case, that datagram has to be sent again.

Note by the way that the terms "datagram" and "packet" often seem to be nearly interchangeable. Technically, datagram is the right word to use when describing TCP/IP. A datagram is a unit of data, which is what the protocols deal with. A packet is a physical thing, appearing on an Ethernet or some wire. In most cases a packet simply contains a datagram, so there is very little difference. However they can differ. When TCP/IP is used on top of X.25, the X.25 interface breaks the datagrams up into 128-byte packets. This is invisible to IP, because the packets are put back together into a single datagram at the other end before being processed by TCP/IP. So in this case, one IP datagram would be carried by several packets. However with most media, there are efficiency advantages to sending one datagram per packet, and so the distinction tends to vanish.

2.1. The TCP level

Two separate protocols are involved in handling TCP/IP datagrams. TCP (the "transmission control protocol") is responsible for breaking up the message into datagrams, reassembling them at the other end, resending anything that gets lost, and putting things back in the right order. IP (the "internet protocol") is responsible for routing individual datagrams. It may seem like TCP is doing all the work. And in small networks that is true. However in the Internet, simply getting a datagram to its destination can be a complex job. A connection may require the datagram to go through several networks at Rutgers, a serial line to the John von Neuman Supercomputer Center, a couple of Ethernets there, a series of 56Kbaud phone lines to another NSFnet site, and more Ethernets on another campus. Keeping track of the routes to all of the destinations and handling incompatibilities among different transport media turns out to be a complex job. Note that the interface between TCP and IP is fairly simple. TCP simply hands IP a datagram with a destination. IP doesn't know how this datagram relates to any datagram before it or after it.

It may have occurred to you that something is missing here. We have talked about Internet addresses, but not about how you keep track of multiple connections to a given system. Clearly it isn't enough to get a datagram to the right destination. TCP has to know which connection this datagram is part of. This task is referred to as "demultiplexing." In fact, there are several levels of demultiplexing going on in TCP/IP. The information needed to do this demultiplexing is contained in a series of "headers". A header is simply a few extra octets tacked onto the beginning of a datagram by some protocol in order to keep track of it. It's a lot like putting a letter into an envelope and putting an address on the outside of the envelope. Except with modern networks it happens several times. It's like you put the letter into a little envelope, your secretary puts that into a somewhat bigger envelope, the campus mail center puts that envelope into a still bigger one, etc. Here is an overview of the headers that get stuck on a message that passes through a typical TCP/IP network:

We start with a single data stream, say a file you are trying to send to some other computer:

```
                    ....................................................
```

TCP breaks it up into manageable chunks. (In order to do this, TCP has to know how large a datagram your network can handle. Actually, the TCP's at each end say how big a datagram they can handle, and then they pick the smallest size.)

```
                    ....   ....   ....   ....   ....   ....   ....   ....
```

TCP puts a header at the front of each datagram. This header actually contains at least 20 octets, but the most important ones are a source and destination "port number" and a "sequence number". The port numbers are used to keep track of different conversations. Suppose 3 different people are transferring files. Your TCP might allocate port numbers 1000, 1001, and 1002 to these transfers. When you are sending a datagram, this becomes the "source" port number, since you are the source of the datagram. Of course the TCP at the other end has assigned a port number of its own for the conversation. Your TCP has to know the port number used by the other end as well. (It finds out when the connection starts, as we will explain below.) It puts this in the "destination" port field. Of course if the other end sends a datagram back to you, the source and destination port numbers will be reversed, since then it will be the source and you will be the destination. Each datagram has a sequence number. This is used so that the other end can make sure that it gets the datagrams in the right order, and that it hasn't missed any. (See the TCP specification for details.) TCP doesn't number the datagrams, but the octets. So if there are 500 octets of data in each datagram, the first datagram might be numbered 0, the second 500, the next 1000, the next 1500, etc. Finally, I will mention the Checksum. This is a number that is computed by adding up all the octets in the datagram (more or less - see the TCP spec). The result is put in the header. TCP at the other end computes the checksum again. If they disagree, then something bad happened to the datagram in transmission, and it is thrown away. So here's what the datagram looks like now.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|                      Sequence Number                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

i+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

i+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

i+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

i+
|   your data ... next 500 octets                               |
|   ......                                                      |
```

If we abbreviate the TCP header as "T", the whole file now looks like this:

T.... T.... T.... T.... T.... T.... T....

You will note that there are items in the header that I have not described above. They are generally involved with managing the connection. In order to make sure the datagram has arrived at its destination, the recipient has to send back an "acknowledgement". This is a datagram whose "Acknowledgement number" field is filled in. For example, sending a packet with an acknowledgement of 1500 indicates that you have received all the data up to octet number 1500. If the sender doesn't get an acknowledgement within a reasonable amount of time, it sends the data again. The window is used to control how much data can be in transit at any one time. It is not practical to wait for each datagram to be acknowledged before sending the next one. That would slow things down too much. On the other hand, you can't just keep sending, or a fast computer might overrun the capacity of a slow one to absorb data. Thus each end indicates how much new data it is currently prepared to absorb by putting the number of octets in its "Window" field. As the computer receives data, the amount of space left in its window decreases. When it goes to zero, the sender has to stop. As the receiver processes the data, it increases its window, indicating that it is ready to accept more data. Often the same datagram can be used to acknowledge receipt of a set of data and to give permission for additional new data (by an updated window). The "Urgent" field allows one end to tell the other to skip ahead in its processing to a particular octet. This is often useful for handling asynchronous events, for example when you type a control character or other command that interrupts output. The other fields are beyond the scope of this document.

2.2. The IP level

TCP sends each of these datagrams to IP. Of course it has to tell IP the Internet address of the computer at the other end. Note that this is all IP is concerned about. It doesn't care about what is in the datagram, or even in the TCP header. IP's job is simply to find a route for the datagram and get it to the other end. In order to allow gateways or other intermediate systems to forward the datagram, it adds its own header. The main things in this header are the source and destination Internet address (32-bit addresses, like 128.6.4.194), the protocol number, and another checksum. The source Internet address is simply the address of your machine. (This is necessary so the other end knows where the datagram came from.) The destination Internet address is the address of the other machine. (This is necessary so any gateways in the middle know where you want the datagram to go.) The protocol number tells IP at the other end to send the datagram to TCP. Although most IP traffic uses TCP, there are other protocols that can use IP, so you have to tell IP which protocol to send the datagram to. Finally, the checksum allows IP at the other end to verify that the header wasn't damaged in transit. Note that TCP and IP have separate checksums. IP needs to be able to verify that the header didn't get damaged in transit, or it could send a message to the wrong place. For reasons not worth discussing here, it is both more efficient and safer to have TCP compute a separate checksum for the TCP header and data. Once IP has tacked on its header, here's what the message looks like:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
i+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|  TCP header, then your data ......                            |
|                                                               |
```

If we represent the IP header by an "I", your file now looks like this:

IT.... IT.... IT.... IT.... IT.... IT.... IT....

Again, the header contains some additional fields that have not been discussed. Most of them are beyond the scope of this document. The flags and fragment offset are used to keep track of the pieces when a datagram has to be split up. This can happen when datagrams are forwarded through a network for which they are too big. (This will be discussed a bit more below.) The time to live is a number that is decremented whenever the datagram passes through a system. When it goes to zero, the datagram is discarded. This is done in case a loop develops in the system somehow. Of course this should be impossible, but well-designed networks are built to cope with "impossible" conditions.

At this point, it's possible that no more headers are needed. If your computer happens to have a direct phone line connecting it to the destination computer, or to a gateway, it may simply send the datagrams out on the line (though likely a synchronous protocol such as HDLC would be used, and it would add at least a few octets at the beginning and end).

2.3. The Ethernet level

However most of our networks these days use Ethernet. So now we have to describe Ethernet's headers. Unfortunately, Ethernet has its own addresses. The people who designed Ethernet wanted to make sure that no two machines would end up with the same Ethernet address. Furthermore, they didn't want the user to have to worry about assigning addresses. So each Ethernet controller comes with an address builtin from the factory. In order to make sure that they would never have to reuse addresses, the Ethernet designers allocated 48 bits for the Ethernet address. People who make Ethernet equipment have to register with a central authority, to make sure that the numbers they assign don't overlap any other manufacturer. Ethernet is a "broadcast medium". That is, it is in effect like an old party line telephone. When you send a packet out on the Ethernet, every machine on the network sees the packet. So something is needed to make sure that the right machine gets it. As you might guess, this involves the Ethernet header. Every Ethernet packet has a 14-octet header that includes the source and destination Ethernet address, and a type code. Each machine is supposed to pay attention only to packets with its own Ethernet address in the destination field. (It's perfectly possible to cheat, which is one reason that Ethernet communications are not terribly secure.) Note that there is no connection between the Ethernet address and the Internet address. Each machine has to have a table of what Ethernet address corresponds to what Internet address. (We will describe how this table is constructed a bit later.) In addition to the addresses, the header contains a type code. The type code is to allow for several different protocol families to be used on the same network. So you can use TCP/IP, DECnet, Xerox NS, etc. at the same time. Each of them will put a different value in the type field. Finally, there is a checksum. The Ethernet controller computes a checksum of the entire packet. When the other end receives the packet, it recomputes the checksum, and throws the packet away if the answer disagrees with the original. The checksum is put on the end of the packet, not in the header. The final result is that your message looks like this:

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|       Ethernet destination address (first 32 bits)            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
| Ethernet dest (last 16 bits)  |Ethernet source (first 16 bits)|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|       Ethernet source address (last 32 bits)                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|       Type code               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|  IP header, then TCP header, then your data                   |
|                                                               |
    ...
|                                                               |
|   end of your data                                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
|                        Ethernet Checksum                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



i+
```

If we represent the Ethernet header with "E", and the Ethernet checksum with "C", your file now looks like this:

EIT....C EIT....C EIT....C EIT....C EIT....C

When these packets are received by the other end, of course all the headers are removed. The Ethernet interface removes the Ethernet header and the checksum. It looks at the type code. Since the type code is the one assigned to IP, the Ethernet device driver passes the datagram up to IP. IP removes the IP header. It looks at the IP protocol field. Since the protocol type is TCP, it passes the datagram up to TCP. TCP now looks at the sequence number. It uses the sequence numbers and other information to combine all the datagrams into the original file.

The ends our initial summary of TCP/IP. There are still some crucial concepts we haven't gotten to, so we'll now go back and add details in several areas. (For detailed descriptions of the items discussed here see, RFC 793 for TCP, RFC 791 for IP, and RFC's 894 and 826 for sending IP over Ethernet.)

3. Well-known sockets and the applications layer

So far, we have described how a stream of data is broken up into datagrams, sent to another computer, and put back together. However something more is needed in order to accomplish anything useful. There has to be a way for you to open a connection to a specified computer, log into it, tell it what file you want, and control the transmission of the file. (If you have a different application in mind, e.g. computer mail, some analogous protocol is needed.) This is done by "application protocols". The application protocols run "on top" of TCP/IP. That is, when they want to send a message, they give the message to TCP. TCP makes sure it gets delivered to the other end. Because TCP and IP take care of all the networking details, the applications protocols can treat a network connection as if it were a simple byte stream, like a terminal or phone line.

Before going into more details about applications programs, we have to describe how you find an application. Suppose you want to send a file to a computer whose Internet address is 128.6.4.7. To start the process, you need more than just the Internet address. You have to connect to the FTP server at the other end. In general, network programs are specialized for a specific set of tasks. Most systems have separate programs to handle file transfers, remote terminal logins, mail, etc. When you connect to 128.6.4.7, you have to specify that you want to talk to the FTP server. This is done by having "well-known sockets" for each server. Recall that TCP uses port numbers to keep track of individual conversations. User programs normally use more or less random port numbers. However specific port numbers are assigned to the programs that sit waiting for requests. For example, if you want to send a file, you will start a program called "ftp". It will open a connection using some random number, say 1234, for the port number on its end. However it will specify port number 21 for the other end. This is the official port number for the FTP server. Note that there are two different programs involved. You run ftp on your side. This is a program designed to accept commands from your terminal and pass them on to the other end. The program that you talk to on the other machine is the FTP server. It is designed to accept commands from the network connection, rather than an interactive terminal. There is no need for your program to use a well-known socket number for itself. Nobody is trying to find it. However the servers have to have well-known numbers, so that people can open connections to them and start sending them commands. The official port numbers for each program are given in "Assigned Numbers".

Note that a connection is actually described by a set of 4 numbers: the Internet address at each end, and the TCP port number at each end. Every datagram has all four of those numbers in it. (The Internet addresses are in the IP header, and the TCP port numbers are in the TCP header.) In order to keep things straight, no two connections can have the same set of numbers. However it is enough for any one number to be different. For example, it is perfectly possible for two different users on a machine to be sending files to the same other machine. This could result in connections with the following parameters:

center,allbox;
c c c.

	Internet addresses	TCP ports
connection 1	128.6.4.194, 128.6.4.7	1234, 21
connection 2	128.6.4.194, 128.6.4.7	1235, 21

Since the same machines are involved, the Internet addresses are the same. Since they are both doing file transfers, one end of the connection involves the well-known port number for FTP. The only thing that differs is the port number for the program that the users are running. That's enough of a difference. Generally, at least one end of the connection asks the network software to assign it a port number that is guaranteed to be unique. Normally, it's the user's end, since the server has to use a well-known number.

Now that we know how to open connections, let's get back to the applications programs. As mentioned earlier, once TCP has opened a connection, we have something that might as well be a simple wire. All the hard parts are handled by TCP and IP. However we still need some agreement as to what we send over this connection. In effect this is simply an agreement on what set of commands the application will understand, and the format in which they are to be sent. Generally, what is sent is a combination of commands and data. They use context to differentiate. For example, the mail protocol works like this: Your mail program opens a connection to the mail server at the other end. Your program gives it your machine's name, the sender of the message, and the recipients you want it sent to. It then sends a command saying that it is starting the message. At that point, the other end stops treating what it sees as commands, and starts accepting the message. Your end then starts sending the text of the message. At the end of the message, a special mark is sent (a dot in the first column). After that, both ends understand that your program is again sending commands. This is the simplest way to do things, and the one that most applications use.

File transfer is somewhat more complex. The file transfer protocol involves two different connections. It starts out just like mail. The user's program sends commands like "log me in as this user", "here is my password", "send me the file with this name". However once the command to send data is sent, a second connection is opened for the data itself. It would certainly be possible to send the data on the same connection, as mail does. However file transfers often take a long time. The designers of the file transfer protocol wanted to allow the user to continue issuing commands while the transfer is going on. For example, the user might make an inquiry, or he might abort the transfer. Thus the designers felt it was best to use a separate connection for the data and leave the original command connection for commands. (It is also possible to open command connections to two different computers, and tell them to send a file from one to the other. In that case, the data couldn't go over the command connection.)

Remote terminal connections use another mechanism still. For remote logins, there is just one connection. It normally sends data. When it is necessary to send a command (e.g. to set the terminal type or to change some mode), a special character is used to indicate that the next character is a command. If the user happens to type that special character as data, two of them are sent.

We are not going to describe the application protocols in detail in this document. It's better to read the RFC's yourself. However there are a couple of common conventions used by applications that will be described here. First, the common network representation: TCP/IP is intended to be usable on any computer. Unfortunately, not all computers agree on how data is represented. There are differences in character codes (ASCII vs. EBCDIC), in end of line conventions (carriage return, line feed, or a representation using counts), and in whether terminals expect characters to be sent individually or a line at a time. In order to allow computers of different kinds to communicate, each applications protocol defines a standard representation. Note that TCP and IP do not care about the representation. TCP simply sends octets. However the programs at both ends have to agree on how the octets are to be interpreted. The RFC for each application specifies the standard representation for that application. Normally it is "net ASCII". This uses ASCII characters, with end of line denoted by a carriage return followed by a line feed. For remote login, there is also a definition of a "standard terminal", which turns out to be a half-duplex terminal with echoing happening on the local machine. Most applications also make provisions for the two computers to agree on other representations that they may find more convenient. For example, PDP-10's have 36-bit words. There is a way that two PDP-10's can agree to send a 36-bit binary file. Similarly, two systems that prefer full-duplex terminal conversations can agree on that. However each application has a standard representation, which every machine must support.

3.1. An example application: SMTP

In order to give a bit better idea what is involved in the application protocols, I'm going to show an example of SMTP, which is the mail protocol. (SMTP is "simple mail transfer protocol.) We assume that a computer called TOPAZ.RUTGERS.EDU wants to send the following message.

```
Date: Sat, 27 Jun 87 13:26:31 EDT
From: hedrick@topaz.rutgers.edu
To: levy@red.rutgers.edu
Subject: meeting

Let's get together Monday at 1pm.
```

First, note that the format of the message itself is described by an Internet standard (RFC 822). The standard specifies the fact that the message must be transmitted as net ASCII (i.e. it must be ASCII, with carriage return/linefeed to delimit lines). It also describes the general structure, as a group of header lines, then a blank line, and then the body of the message. Finally, it describes the syntax of the header lines in detail. Generally they consist of a keyword and then a value.

Note that the addressee is indicated as LEVY@RED.RUTGERS.EDU. Initially, addresses were simply "person at machine". However recent standards have made things more flexible. There are now provisions for systems to handle other systems' mail. This can allow automatic forwarding on behalf of computers not connected to the Internet. It can be used to direct mail for a number of systems to one central mail server. Indeed there is no requirement that an actual computer by the name of RED.RUTGERS.EDU even exist. The name servers could be set up so that you mail to department names, and each department's mail is routed automatically to an appropriate computer. It is also possible that the part before the @ is something other than a user name. It is possible for programs to be set up to process mail. There are also provisions to handle mailing lists, and generic names such as "postmaster" or "operator".

The way the message is to be sent to another system is described by RFC's 821 and 974. The program that is going to be doing the sending asks the name server several queries to determine where to route the message. The first query is to find out which machines handle mail for the name RED.RUTGERS.EDU. In this case, the server replies that RED.RUTGERS.EDU handles its own mail. The program then asks for the address of RED.RUTGERS.EDU, which is 128.6.4.2. Then the mail program opens a TCP connection to port 25 on 128.6.4.2. Port 25 is the well-known socket used for receiving mail. Once this connection is established, the mail program starts sending commands. Here is a typical conversation. Each line is labelled as to whether it is from TOPAZ or RED. Note that TOPAZ initiated the connection:

```
RED    220 RED.RUTGERS.EDU SMTP Service at 29 Jun 87 05:17:18 EDT
TOPAZ  HELO topaz.rutgers.edu
RED    250 RED.RUTGERS.EDU - Hello, TOPAZ.RUTGERS.EDU
TOPAZ  MAIL From:<hedrick@topaz.rutgers.edu>
RED    250 MAIL accepted
TOPAZ  RCPT To:<levy@red.rutgers.edu>
RED    250 Recipient accepted
TOPAZ  DATA
RED    354 Start mail input; end with <CRLF>.<CRLF>
TOPAZ  Date: Sat, 27 Jun 87 13:26:31 EDT
TOPAZ  From: hedrick@topaz.rutgers.edu
TOPAZ  To: levy@red.rutgers.edu
TOPAZ  Subject: meeting
TOPAZ
TOPAZ  Let's get together Monday at 1pm.
TOPAZ  .
RED    250 OK
TOPAZ  QUIT
RED    221 RED.RUTGERS.EDU Service closing transmission channel
```

First, note that commands all use normal text. This is typical of the Internet standards. Many of the protocols use standard ASCII commands. This makes it easy to watch what is going on and to diagnose problems. For example, the mail program keeps a log of each conversation. If something goes wrong, the log file can simply be mailed to the postmaster. Since it is normal text, he can see what was going on. It also allows a human to interact directly with the mail server, for testing. (Some newer protocols are complex enough that this is not practical. The commands would have to have a syntax that would require a significant parser. Thus there is a tendency for newer protocols to use binary formats. Generally they are structured like C or Pascal record structures.) Second, note that the responses all begin with numbers. This is also typical of Internet protocols. The allowable responses are defined in the protocol. The numbers allow the user program to respond unambiguously. The rest of the response is text, which is normally for use by any human who may be watching or looking at a log. It has no effect on the operation of the programs. (However there is one point at which the protocol uses part of the text of the response.) The commands themselves simply allow the mail program on one end to tell the mail server the information it needs to know in order to deliver the message. In this case, the mail server could get the information by looking at the message itself. But for more complex cases, that would not be safe. Every session must begin with a HELO, which gives the name of the system that initiated the connection. Then the sender and recipients are specified. (There can be more than one RCPT command, if there are several recipients.) Finally the data itself is sent. Note that the text of the message is terminated by a line containing just a period. (If such a line appears in the message, the period is doubled.) After the message is accepted, the sender can send another message, or terminate the session as in the example above.

Generally, there is a pattern to the response numbers. The protocol defines the specific set of responses that can be sent as answers to any given command. However programs that don't want to analyze them in detail can just look at the first digit. In general, responses that begin with a 2 indicate success. Those that begin with 3 indicate that some further action is needed, as shown above. 4 and 5 indicate errors. 4 is a "temporary" error, such as a disk filling. The message should be saved, and tried again later. 5 is a permanent error, such as a non-existent recipient. The message should be returned to the sender with an error message.

(For more details about the protocols mentioned in this section, see RFC's 821/822 for mail, RFC 959 for file transfer, and RFC's 854/855 for remote logins. For the well-known port numbers, see the current edition of Assigned Numbers, and possibly RFC 814.)

4. Protocols other than TCP: UDP and ICMP

So far, we have described only connections that use TCP. Recall that TCP is responsible for breaking up messages into datagrams, and reassembling them properly. However in many applications, we have messages that will always fit in a single datagram. An example is name lookup. When a user attempts to make a connection to another system, he will generally specify the system by name, rather than Internet address. His system has to translate that name to an address before it can do anything. Generally, only a few systems have the database used to translate names to addresses. So the user's system will want to send a query to one of the systems that has the database. This query is going to be very short. It will certainly fit in one datagram. So will the answer. Thus it seems silly to use TCP. Of course TCP does more than just break things up into datagrams. It also makes sure that the data arrives, resending datagrams where necessary. But for a question that fits in a single datagram, we don't need all the complexity of TCP to do this. If we don't get an answer after a few seconds, we can just ask again. For applications like this, there are alternatives to TCP.

The most common alternative is UDP ("user datagram protocol"). UDP is designed for applications where you don't need to put sequences of datagrams together. It fits into the system much like TCP. There is a UDP header. The network software puts the UDP header on the front of your data, just as it would put a TCP header on the front of your data. Then UDP sends the data to IP, which adds the IP header, putting UDP's protocol number in the protocol field instead of TCP's protocol number. However UDP doesn't do as much as TCP does. It doesn't split data into multiple datagrams. It doesn't keep track of what it has sent so it can resend if necessary. About all that UDP provides is port numbers, so that several programs can use UDP at once. UDP port numbers are used just like TCP port numbers. There are well-known port numbers for servers that use UDP. Note that the UDP header is shorter than a TCP header. It still has source and destination port numbers, and a checksum, but that's about it. No sequence number, since it is not needed. UDP is used by the protocols that handle name lookups (see IEN 116, RFC 882, and RFC 883), and a number of similar protocols.

Another alternative protocol is ICMP ("Internet control message protocol"). ICMP is used for error messages, and other messages intended for the TCP/IP software itself, rather than any particular user program. For example, if you attempt to connect to a host, your system may get back an ICMP message saying "host unreachable". ICMP can also be used to find out some information about the network. See RFC 792 for details of ICMP. ICMP is similar to UDP, in that it handles messages that fit in one datagram. However it is even simpler than UDP. It doesn't even have port numbers in its header. Since all ICMP messages are interpreted by the network software itself, no port numbers are needed to say where a ICMP message is supposed to go.

5. Keeping track of names and information: the domain system

As we indicated earlier, the network software generally needs a 32-bit Internet address in order to open a connection or send a datagram. However users prefer to deal with computer names rather than numbers. Thus there is a database that allows the software to look up a name and find the corresponding number. When the Internet was small, this was easy. Each system would have a file that listed all of the other systems, giving both their name and number. There are now too many computers for this approach to be practical. Thus these files have been replaced by a set of name servers that keep track of host names and the corresponding Internet addresses. (In fact these servers are somewhat more general than that. This is just one kind of information stored in the domain system.) Note that a set of interlocking servers are used, rather than a single central one. There are now so many different institutions connected to the Internet that it would be impractical for them to notify a central authority whenever they installed or moved a computer. Thus naming authority is delegated to individual institutions. The name servers form a tree, corresponding to institutional structure. The names themselves follow a similar structure. A typical example is the name BORAX.LCS.MIT.EDU. This is a computer at the Laboratory for Computer Science (LCS) at MIT. In order to find its Internet address, you might potentially have to consult 4 different servers. First, you would ask a central server (called the root) where the EDU server is. EDU is a server that keeps track of educational institutions. The root server would give you the names and Internet addresses of several servers for EDU. (There are several servers at each level, to allow for the possibly that one might be down.) You would then ask EDU where the server for MIT is. Again, it would give you names and Internet addresses of several servers for MIT. Generally, not all of those servers would be at MIT, to allow for the possibility of a general power failure at MIT. Then you would ask MIT where the server for LCS is, and finally you would ask one of the LCS servers about BORAX. The final result would be the Internet address for BORAX.LCS.MIT.EDU. Each of these levels is referred to as a "domain". The entire name, BORAX.LCS.MIT.EDU, is called a "domain name". (So are the names of the higher-level domains, such as LCS.MIT.EDU, MIT.EDU, and EDU.)

Fortunately, you don't really have to go through all of this most of the time. First of all, the root name servers also happen to be the name servers for the top-level domains such as EDU. Thus a single query to a root server will get you to MIT. Second, software generally remembers answers that it got before. So once we look up a name at LCS.MIT.EDU, our software remembers where to find servers for LCS.MIT.EDU, MIT.EDU, and EDU. It also remembers the translation of BORAX.LCS.MIT.EDU. Each of these pieces of information has a "time to live" associated with it. Typically this is a few days. After that, the information expires and has to be looked up again. This allows institutions to change things.

The domain system is not limited to finding out Internet addresses. Each domain name is a node in a database. The node can have records that define a number of different properties. Examples are Internet address, computer type, and a list of services provided by a computer. A program can ask for a specific piece of information, or all information about a given name. It is possible for a node in the database to be marked as an "alias" (or nickname) for another node. It is also possible to use the domain system to store information about users, mailing lists, or other objects.

There is an Internet standard defining the operation of these databases, as well as the protocols used to make queries of them. Every network utility has to be able to make such queries, since this is now the official way to evaluate host names. Generally utilities will talk to a server on their own system. This server will take care of contacting the other servers for them. This keeps down the amount of code that has to be in each application program.

The domain system is particularly important for handling computer mail. There are entry types to define what computer handles mail for a given name, to specify where an individual is to receive mail, and to define mailing lists.

(See RFC's 882, 883, and 973 for specifications of the domain system. RFC 974 defines the use of the domain system in sending mail.)

6. Routing

The description above indicated that the IP implementation is responsible for getting datagrams to the destination indicated by the destination address, but little was said about how this would be done. The task of finding how to get a datagram to its destination is referred to as "routing". In fact many of the details depend upon the particular implementation. However some general things can be said.

First, it is necessary to understand the model on which IP is based. IP assumes that a system is attached to some local network. We assume that the system can send datagrams to any other system on its own network. (In the case of Ethernet, it simply finds the Ethernet address of the destination system, and puts the datagram out on the Ethernet.) The problem comes when a system is asked to send a datagram to a system on a different network. This problem is handled by gateways. A gateway is a system that connects a network with one or more other networks. Gateways are often normal computers that happen to have more than one network interface. For example, we have a Unix machine that has two different Ethernet interfaces. Thus it is connected to networks 128.6.4 and 128.6.3. This machine can act as a gateway between those two networks. The software on that machine must be set up so that it will forward datagrams from one network to the other. That is, if a machine on network 128.6.4 sends a datagram to the gateway, and the datagram is addressed to a machine on network 128.6.3, the gateway will forward the datagram to the destination. Major communications centers often have gateways that connect a number of different networks. (In many cases, special-purpose gateway systems provide better performance or reliability than general-purpose systems acting as gateways. A number of vendors sell such systems.)

Routing in IP is based entirely upon the network number of the destination address. Each computer has a table of network numbers. For each network number, a gateway is listed. This is the gateway to be used to get to that network. Note that the gateway doesn't have to connect directly to the network. It just has to be the best place to go to get there. For example at Rutgers, our interface to NSFnet is at the John von Neuman Supercomputer Center

(JvNC). Our connection to JvNC is via a high-speed serial line connected to a gateway whose address is 128.6.3.12. Systems on net 128.6.3 will list 128.6.3.12 as the gateway for many off-campus networks. However systems on net 128.6.4 will list 128.6.4.1 as the gateway to those same off-campus networks. 128.6.4.1 is the gateway between networks 128.6.4 and 128.6.3, so it is the first step in getting to JvNC.

When a computer wants to send a datagram, it first checks to see if the destination address is on the system's own local network. If so, the datagram can be sent directly. Otherwise, the system expects to find an entry for the network that the destination address is on. The datagram is sent to the gateway listed in that entry. This table can get quite big. For example, the Internet now includes several hundred individual networks. Thus various strategies have been developed to reduce the size of the routing table. One strategy is to depend upon "default routes". Often, there is only one gateway out of a network. This gateway might connect a local Ethernet to a campus-wide backbone network. In that case, we don't need to have a separate entry for every network in the world. We simply define that gateway as a "default". When no specific route is found for a datagram, the datagram is sent to the default gateway. A default gateway can even be used when there are several gateways on a network. There are provisions for gateways to send a message saying "I'm not the best gateway -- use this one instead." (The message is sent via ICMP. See RFC 792.) Most network software is designed to use these messages to add entries to their routing tables. Suppose network 128.6.4 has two gateways, 128.6.4.59 and 128.6.4.1. 128.6.4.59 leads to several other internal Rutgers networks. 128.6.4.1 leads indirectly to the NSFnet. Suppose we set 128.6.4.59 as a default gateway, and have no other routing table entries. Now what happens when we need to send a datagram to MIT? MIT is network 18. Since we have no entry for network 18, the datagram will be sent to the default, 128.6.4.59. As it happens, this gateway is the wrong one. So it will forward the datagram to 128.6.4.1. But it will also send back an error saying in effect: "to get to network 18, use 128.6.4.1". Our software will then add an entry to the routing table. Any future datagrams to MIT will then go directly to 128.6.4.1. (The error message is sent using the ICMP protocol. The message type is called "ICMP redirect.")

Most IP experts recommend that individual computers should not try to keep track of the entire network. Instead, they should start with default gateways, and let the gateways tell them the routes, as just described. However this doesn't say how the gateways should find out about the routes. The gateways can't depend upon this strategy. They have to have fairly complete routing tables. For this, some sort of routing protocol is needed. A routing protocol is simply a technique for the gateways to find each other, and keep up to date about the best way to get to every network. RFC 1009 contains a review of gateway design and routing. However rip.doc is probably a better introduction to the subject. It contains some tutorial material, and a detailed description of the most commonly-used routing protocol.

7. Details about Internet addresses: subnets and broadcasting

As indicated earlier, Internet addresses are 32-bit numbers, normally written as 4 octets (in decimal), e.g. 128.6.4.7. There are actually 3 different types of address. The problem is that the address has to indicate both the network and the host within the network. It was felt that eventually there would be lots of networks. Many of them would be small, but probably 24 bits would be needed to represent all the IP networks. It was also felt that some very big networks might need 24 bits to represent all of their hosts. This would seem to lead to 48 bit addresses. But the designers really wanted to use 32 bit addresses. So they adopted a kludge. The assumption is that most of the networks will be small. So they set up three different ranges of address. Addresses beginning with 1 to 126 use only the first octet for the network number. The other three octets are available for the host number. Thus 24 bits are available for hosts. These numbers are used for large networks. But there can only be 126 of these very big networks. The Arpanet is one, and there are a few large commercial networks. But few normal organizations get one of these "class A" addresses. For normal large organizations, "class B" addresses are used. Class B addresses use the first two octets for the network number. Thus network numbers are 128.1 through 191.254. (We avoid 0 and 255, for reasons that we see below. We also avoid addresses beginning with 127, because that is used by some systems for special purposes.) The last two octets are available for host addesses, giving 16 bits of host address. This allows for 64516 computers, which should be enough for most organizations. (It is possible to get more than one class B address, if you run out.) Finally, class C addresses use three octets, in the range 192.1.1 to 223.254.254. These allow only 254 hosts on each network, but there can be lots of these networks. Addresses above 223 are reserved for future use, as class D and E (which are currently not defined).

Many large organizations find it convenient to divide their network number into "subnets". For example, Rutgers has been assigned a class B address, 128.6. We find it convenient to use the third octet of the address to indicate which Ethernet a host is on. This division has no significance outside of Rutgers. A computer at another institution would treat all datagrams addressed to 128.6 the same way. They would not look at the third octet of the address. Thus computers outside Rutgers would not have different routes for 128.6.4 or 128.6.5. But inside Rutgers, we treat 128.6.4 and 128.6.5 as separate networks. In effect, gateways inside Rutgers have separate entries for each Rutgers subnet, whereas gateways outside Rutgers just have one entry for 128.6. Note that we could do exactly the same thing by using a separate class C address for each Ethernet. As far as Rutgers is concerned, it would be just as convenient for us to have a number of class C addresses. However using class C addresses would make things inconvenient for the rest of the world. Every institution that wanted to talk to us would have to have a separate entry for each one of our networks. If every institution did this, there would be far too many networks for any reasonable gateway to keep track of. By subdividing a class B network, we hide our internal structure from everyone else, and save them trouble. This subnet strategy requires special provisions in the network software. It is described in RFC 950.

0 and 255 have special meanings. 0 is reserved for machines that don't know their address. In certain circumstances it is possible for a machine not to know the number of the network it is on, or even its own host address. For example, 0.0.0.23 would be a machine that knew it was host number 23, but didn't know on what network.

255 is used for "broadcast". A broadcast is a message that you want every system on the network to see. Broadcasts are used in some situations where you don't know who to talk to. For example, suppose you need to look up a host name and get its Internet address. Sometimes you don't know the address of the nearest name server. In that case, you might send the request as a broadcast. There are also cases where a number of systems are interested in information. It is then less expensive to send a single broadcast than to send datagrams

individually to each host that is interested in the information. In order to send a broadcast, you use an address that is made by using your network address, with all ones in the part of the address where the host number goes. For example, if you are on network 128.6.4, you would use 128.6.4.255 for broadcasts. How this is actually implemented depends upon the medium. It is not possible to send broadcasts on the Arpanet, or on point to point lines. However it is possible on an Ethernet. If you use an Ethernet address with all its bits on (all ones), every machine on the Ethernet is supposed to look at that datagram.

Although the official broadcast address for network 128.6.4 is now 128.6.4.255, there are some other addresses that may be treated as broadcasts by certain implementations. For convenience, the standard also allows 255.255.255.255 to be used. This refers to all hosts on the local network. It is often simpler to use 255.255.255.255 instead of finding out the network number for the local network and forming a broadcast address such as 128.6.4.255. In addition, certain older implementations may use 0 instead of 255 to form the broadcast address. Such implementations would use 128.6.4.0 instead of 128.6.4.255 as the broadcast address on network 128.6.4. Finally, certain older implementations may not understand about subnets. Thus they consider the network number to be 128.6. In that case, they will assume a broadcast address of 128.6.255.255 or 128.6.0.0. Until support for broadcasts is implemented properly, it can be a somewhat dangerous feature to use.

Because 0 and 255 are used for unknown and broadcast addresses, normal hosts should never be given addresses containing 0 or 255. Addresses should never begin with 0, 127, or any number above 223. Addresses violating these rules are sometimes referred to as "Martians", because of rumors that the Central University of Mars is using network 225.

8. Datagram fragmentation and reassembly

TCP/IP is designed for use with many different kinds of network. Unfortunately, network designers do not agree about how big packets can be. Ethernet packets can be 1500 octets long. Arpanet packets have a maximum of around 1000 octets. Some very fast networks have much larger packet sizes. At first, you might think that IP should simply settle on the smallest possible size. Unfortunately, this would cause serious performance problems. When transferring large files, big packets are far more efficient than small ones. So we want to be able to use the largest packet size possible. But we also want to be able to handle networks with small limits. There are two provisions for this. First, TCP has the ability to "negotiate" about datagram size. When a TCP connection first opens, both ends can send the maximum datagram size they can handle. The smaller of these numbers is used for the rest of the connection. This allows two implementations that can handle big datagrams to use them, but also lets them talk to implementations that can't handle them. However this doesn't completely solve the problem. The most serious problem is that the two ends don't necessarily know about all of the steps in between. For example, when sending data between Rutgers and Berkeley, it is likely that both computers will be on Ethernets. Thus they will both be prepared to handle 1500-octet datagrams. However the connection will at some point end up going over the Arpanet. It can't handle packets of that size. For this reason, there are provisions to split datagrams up into pieces. (This is referred to as "fragmentation".) The IP header contains fields indicating the a datagram has been split, and enough information to let the pieces be put back together. If a gateway connects an Ethernet to the Arpanet, it must be prepared to take 1500-octet Ethernet packets and split them into pieces that will fit on the Arpanet. Furthermore, every host implementation of TCP/IP must be prepared to accept pieces and put them back together. This is referred to as "reassembly".

TCP/IP implementations differ in the approach they take to deciding on datagram size. It is fairly common for implementations to use 576-byte datagrams whenever they can't verify that the entire path is able to handle larger packets. This rather conservative strategy is used because of the number of implementations with bugs in the code to reassemble fragments. Implementors often try to avoid ever having fragmentation occur. Different implementors take different approaches to deciding when it is safe to use large datagrams. Some use them only for the local network. Others will use them for any network on the same campus. 576 bytes is a "safe" size, which every implementation must support.

9. Ethernet encapsulation: ARP

There was a brief discussion earlier about what IP datagrams look like on an Ethernet. The discussion showed the Ethernet header and checksum. However it left one hole: It didn't say how to figure out what Ethernet address to use when you want to talk to a given Internet address. In fact, there is a separate protocol for this, called ARP ("address resolution protocol"). (Note by the way that ARP is not an IP protocol. That is, the ARP datagrams do not have IP headers.) Suppose you are on system 128.6.4.194 and you want to connect to system 128.6.4.7. Your system will first verify that 128.6.4.7 is on the same network, so it can talk directly via Ethernet. Then it will look up 128.6.4.7 in its ARP table, to see if it already knows the Ethernet address. If so, it will stick on an Ethernet header, and send the packet. But suppose this system is not in the ARP table. There is no way to send the packet, because you need the Ethernet address. So it uses the ARP protocol to send an ARP request. Essentially an ARP request says "I need the Ethernet address for 128.6.4.7". Every system listens to ARP requests. When a system sees an ARP request for itself, it is required to respond. So 128.6.4.7 will see the request, and will respond with an ARP reply saying in effect "128.6.4.7 is 8:0:20:1:56:34". (Recall that Ethernet addresses are 48 bits. This is 6 octets. Ethernet addresses are conventionally shown in hex, using the punctuation shown.) Your system will save this information in its ARP table, so future packets will go directly. Most systems treat the ARP table as a cache, and clear entries in it if they have not been used in a certain period of time.

Note by the way that ARP requests must be sent as "broadcasts". There is no way that an ARP request can be sent directly to the right system. After all, the whole reason for sending an ARP request is that you don't know the Ethernet address. So an Ethernet address of all ones is used, i.e. ff:ff:ff:ff:ff:ff. By convention, every machine on the Ethernet is required to pay attention to packets with this as an address. So every system sees every ARP requests. They all look to see whether the request is for their own address. If so, they respond. If not, they could just ignore it. (Some hosts will use ARP requests to update their knowledge about other hosts on the network, even if the request isn't for them.) Note that packets whose IP address indicates broadcast (e.g. 255.255.255.255 or 128.6.4.255) are also sent with an Ethernet address that is all ones.

10. Getting more information

This directory contains documents describing the major protocols. There are literally hundreds of documents, so we have chosen the ones that seem most important. Internet standards are called RFC's. RFC stands for Request for Comment. A proposed standard is initially issued as a proposal, and given an RFC number. When it is finally accepted, it is added to Official Internet Protocols, but it is still referred to by the RFC number. We have also included two IEN's. (IEN's used to be a separate classification for more informal documents. This classification no longer exists -- RFC's are now used for all official Internet documents, and a mailing list is used for more informal reports.) The convention is that whenever an RFC is revised, the revised version gets a new number. This is fine for most purposes, but it causes problems with two documents: Assigned Numbers and Official Internet Protocols. These documents are being revised all the time, so the RFC number keeps changing. You will have to look in rfc-index.txt to find the number of the latest edition. Anyone who is seriously interested in TCP/IP should read the RFC describing IP (791). RFC 1009 is also useful. It is a specification for gateways to be used by NSFnet. As such, it contains an overview of a lot of the TCP/IP technology. You should probably also read the description of at least one of the application protocols, just to get a feel for the way things work. Mail is probably a good one (821/822). TCP (793) is of course a very basic specification. However the spec is fairly complex, so you should only read this when you have the time and patience to think about it carefully. Fortunately, the author of the major RFC's (Jon Postel) is a very good writer. The TCP RFC is far easier to read than you would expect, given the complexity of what it is describing. You can look at the other RFC's as you become curious about their subject matter.

Here is a list of the documents you are more likely to want:

rfc-index	list of all RFC's
rfc1012	somewhat fuller list of all RFC's
rfc1011	Official Protocols. It's useful to scan this to see what tasks protocols have been built for. This defines which RFC's are actual standards, as opposed to requests for comments.
rfc1010	Assigned Numbers. If you are working with TCP/IP, you will probably want a hardcopy of this as a reference. It's not very exciting to read. It lists all the offically defined well-known ports and lots of other things.
rfc1009	NSFnet gateway specifications. A good overview of IP routing and gateway technology.
rfc1001/2	netBIOS: networking for PC's
rfc973	update on domains
rfc959	FTP (file transfer)
rfc950	subnets
rfc937	POP2: protocol for reading mail on PC's
rfc894	how IP is to be put on Ethernet, see also rfc825
rfc882/3	domains (the database used to go from host names to Internet address and back -- also used to handle UUCP these days). See also rfc973
rfc854/5	telnet - protocol for remote logins
rfc826	ARP - protocol for finding out Ethernet addresses
rfc821/2	mail
rfc814	names and ports - general concepts behind well-known ports
rfc793	TCP
rfc792	ICMP
rfc791	IP
rfc768	UDP
rip.doc	details of the most commonly-used routing protocol
ien-116	old name server (still needed by several kinds of system)
ien-48	the Catenet model, general description of the philosophy behind TCP/IP

The following documents are somewhat more specialized.

rfc813	window and acknowledgement strategies in TCP
rfc815	datagram reassembly techniques
rfc816	fault isolation and resolution techniques
rfc817	modularity and efficiency in implementation
rfc879	the maximum segment size option in TCP
rfc896	congestion control
rfc827,888,904,975,985	EGP and related issues

To those of you who may be reading this document remotely instead of at Rutgers: The most important RFC's have been collected into a three-volume set, the DDN Protocol Handbook. It is available from the DDN Network Information Center, SRI International, 333 Ravenswood Avenue, Menlo Park, California 94025 (telephone: 800-235-3155). You should be able to get them via anonymous FTP from sri-nic.arpa. File names are:

```
RFC's:
  rfc:rfc-index.txt
  rfc:rfcxxx.txt
IEN's:
  ien:ien-index.txt
  ien:ien-xxx.txt
```

rip.doc is available by anonymous FTP from topaz.rutgers.edu, as /pub/tcp-ip-docs/rip.doc.

Sites with access to UUCP but not FTP may be able to retreive them via UUCP from UUCP host rutgers. The file names would be

```
RFC's:
  /topaz/pub/pub/tcp-ip-docs/rfc-index.txt
  /topaz/pub/pub/tcp-ip-docs/rfcxxx.txt
IEN's:
  /topaz/pub/pub/tcp-ip-docs/ien-index.txt
  /topaz/pub/pub/tcp-ip-docs/ien-xxx.txt
/topaz/pub/pub/tcp-ip-docs/rip.doc
```

Note that SRI-NIC has the entire set of RFC's and IEN's, but rutgers and topaz have only those specifically mentioned above.

Appendix E – Networking Implementation Notes

The following appendix contains a document produced by Samuel J. Leffler, William N. Joy, Robert S. Fabry, and Michael J. Karels of the Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science at the University of California, Berkeley, titled "*Networking Implementation Notes, 4.3BSD Edition*". The document describes the internal structure of the networking facilities developed at U.C. Berkeley for the 4.3BSD version of the UNIX® operating system.

Networking Implementation Notes
4.3BSD Edition

Samuel J. Leffler, William N. Joy, Robert S. Fabry, and Michael J. Karels

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

This report describes the internal structure of the networking facilities developed for the 4.3BSD version of the UNIX* operating system for the VAX†. These facilities are based on several central abstractions which structure the external (user) view of network communication as well as the internal (system) implementation.

The report documents the internal structure of the networking system. The "Berkeley Software Architecture Manual, 4.3BSD Edition" (PS1:6) provides a description of the user interface to the networking facilities.

Revised June 5, 1986

* UNIX is a trademark of Bell Laboratories.
† DEC, VAX, DECnet, and UNIBUS are trademarks of Digital Equipment Corporation.

TABLE OF CONTENTS

1. Introduction

This report describes the internal structure of facilities added to the 4.2BSD version of the UNIX operating system for the VAX, as modified in the 4.3BSD release. The system facilities provide a uniform user interface to networking within UNIX. In addition, the implementation introduces a structure for network communications which may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *Berkeley Software Architecture Manual, 4.3BSD Edition* [Joy86]. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilized only by the interprocess communication facilities.

2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

4. Internal address representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
struct sockaddr {
        short       sa_family;        /* data format identifier */
        char        sa_data[14];      /* address */
};
```

All addresses belong to one or more ***address families*** which define their format and interpretation. The *sa_family* field indicates the address family to which the address belongs, and the *sa_data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats.* Specific address formats use private structure definitions that define the format of the data field. The system interface supports larger address structures, although address-family-independent support facilities, for example routing and raw socket interfaces, provide only 14 bytes for address storage. Protocols that do not use those facilities (e.g, the current Unix domain) may use larger data areas.

5. Memory management

A single mechanism is used for data storage: memory buffers, or *mbuf*s. An mbuf is a structure of the form:

```
struct mbuf {
        struct      mbuf *m_next;     /* next buffer in chain */
        u_long      m_off;            /* offset of data */
        short       m_len;            /* amount of data in this mbuf */
        short       m_type;           /* mbuf type (accounting) */
        u_char      m_dat[MLEN];      /* data storage */
        struct      mbuf *m_act;      /* link in higher-level mbuf list */
};
```

The *m_next* field is used to chain mbufs together on linked lists, while the *m_act* field allows lists of mbuf chains to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat.* The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define  mtod(x,t)          ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, which is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. An mbuf with an external data area may be recognized by the larger offset to the data area; this is formalized by the macro M_HASCL(*m*), which is true if the mbuf whose address is *m* has an external page cluster. An array of reference counts on pages is also maintained so that copies of pages may be made

* Later versions of the system may support variable length addresses.

without core to core copying (copies are created simply by duplicating the reference to the data and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware whether data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following may be used to allocate and free mbufs:

m = m_get(wait, type);
MGET(m, wait, type);

> The subroutine *m_get* and the macro *MGET* each allocate an mbuf, placing its address in *m*. The argument *wait* is either M_WAIT or M_DONTWAIT according to whether allocation should block or fail if no mbuf is available. The *type* is one of the predefined mbuf types for use in accounting of mbuf allocation.

MCLGET(m);

> This macro attempts to allocate an mbuf page cluster to associate with the mbuf *m*. If successful, the length of the mbuf is set to CLSIZE, the size of the page cluster.

n = m_free(m);
MFREE(m,n);

> The routine *m_free* and the macro *MFREE* each free a single mbuf, *m*, and any associated external storage area, placing a pointer to its successor in the chain it heads, if any, in *n*.

m_freem(m);

> This routine frees an mbuf chain headed by *m*.

The following utility routines are available for manipulating mbuf chains:

m = m_copy(m0, off, len);

> The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M_COPYALL, all the data present, offset as before, is copied.

m_cat(m, n);

> The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

m_adj(m, diff);

> The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation; alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

m = m_pullup(m0, size);

> After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is guaranteed to have at least *size* bytes of data in contiguous memory within the data area of the mbuf (allowing access via a pointer, obtained using the *mtod* macro, and allowing the mbuf to be located from a pointer to the data area using *dtom*, defined below). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.
>
> This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring that mbufs always reside on 128 byte boundaries, it is always possible to locate the mbuf associated with a data area by masking off the low bits of the virtual address.

This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. Note that this works only with objects stored in the internal data buffer of the mbuf. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define     dtom(x)     ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets as well as memory allocated for packets and headers. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat*(1) program.

6. Internal layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces to which each must conform.

6.1. Socket layer

The socket layer deals with the interprocess communication facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. The system calls described in the *Berkeley Software Architecture Manual* [Joy86] are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
        short       so_type;            /* generic type */
        short       so_options;         /* from socket call */
        short       so_linger;          /* time to linger while closing */
        short       so_state;           /* internal state flags */
        caddr_t     so_pcb;             /* protocol control block */
        struct      protosw *so_proto;  /* protocol handle */
        struct      socket *so_head;    /* back pointer to accept socket */
        struct      socket *so_q0;      /* queue of partial connections */
        short       so_q0len;           /* partials on so_q0 */
        struct      socket *so_q;       /* queue of incoming connections */
        short       so_qlen;            /* number of connections on so_q */
        short       so_qlimit;          /* max number queued connections */
        struct      sockbuf so_rcv;     /* receive queue */
        struct      sockbuf so_snd;     /* send queue */
        short       so_timeo;           /* connection timeout */
        u_short     so_error;           /* error affecting connection */
        u_short     so_oobmark;         /* chars to oob mark */
        short       so_pgrp;            /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol-specific data structure, the "protocol control block," is also present in the socket structure. Protocols control this data structure, which normally includes a back pointer to the parent socket structure to allow easy lookup when returning information to a user (for example, placing an error number in the *so_error* field). The other entries in the socket structure are used in queuing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as a "wait channel' to be used in notification. When data arrives for the process and is placed in the socket's queue, the blocked process is identified by the fact it is waiting "on the queue."

6.1.1. Socket state

A socket's state is defined from the following:

```
#define SS_NOFDREF           0x001    /* no file table ref any more */
#define SS_ISCONNECTED       0x002    /* socket connected to a peer */
#define SS_ISCONNECTING      0x004    /* in process of connecting to peer */
#define SS_ISDISCONNECTING   0x008    /* in process of disconnecting */
#define SS_CANTSENDMORE      0x010    /* can't send more data to peer */
#define SS_CANTRCVMORE       0x020    /* can't receive more data from peer */
#define SS_RCVATMARK         0x040    /* at mark on input */

#define SS_PRIV              0x080    /* privileged */
#define SS_NBIO              0x100    /* non-blocking ops */
#define SS_ASYNC             0x200    /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created, the state is defined based on the type of socket. It may change as control actions are performed, for example connection establishment. It may also change according to the type of input/output the user wishes to perform, as indicated by options set with *fcntl*. "Non-blocking" I/O implies that a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULDBLOCK, or the service request may be partially fulfilled, e.g. a request for more data than is present.

If a process requested "asynchronous" notification of events related to the socket, the SIGIO signal is posted to the process when such events occur. An event is a change in the socket's state; examples of such occurrences are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "privileged" if it was created by the super-user. Only privileged sockets may bind addresses in privileged portions of an address space or use "raw" sockets to access lower levels of the network.

6.1.2. Socket data queues

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
struct sockbuf {
        u_short   sb_cc;          /* actual chars in buffer */
        u_short   sb_hiwat;       /* max actual char count */
        u_short   sb_mbcnt;       /* chars of mbufs used */
        u_short   sb_mbmax;       /* max chars of mbufs to use */
        u_short   sb_lowat;       /* low water mark */
        short     sb_timeo;       /* timeout */
        struct    mbuf *sb_mb;    /* the mbuf chain */
        struct    proc *sb_sel;   /* process selecting read/write */
        short     sb_flags;       /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of data characters as well as high and low water marks are used by the protocols in controlling the flow of data. The amount of buffer space (characters of mbufs and associated data pages) is also recorded along with the limit on buffer allocation. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).*

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The limit on buffer allocation is set somewhat higher than the limit on data characters to account for the granularity of buffer allocation. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but it is assumed that this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

Data queued at a socket is stored in one of two styles. Stream-oriented sockets queue data with no addresses, headers or record boundaries. The data are in mbufs linked through the *m_next* field. Buffers containing access rights may be present within the chain if the underlying protocol supports passage of access rights. Record-oriented sockets, including datagram sockets, queue data as a list of packets; the sections of packets are distinguished by the types of the mbufs containing them. The mbufs which comprise a record are linked through the *m_next* field; records are linked from the *m_act* field of the first mbuf of one packet to the first mbuf of the next. Each packet begins with an mbuf containing the "from" address if the protocol provides it, then any buffers containing access rights, and finally any buffers containing data. If a record contains no data, no data buffers are required unless neither address nor access rights are present.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources:

```
#define  SB_LOCK        0x01   /* lock on data queue (so_rcv only) */
#define  SB_WANT        0x02   /* someone is waiting to lock */
#define  SB_WAIT        0x04   /* someone is waiting for data/space */
#define  SB_SEL         0x08   /* buffer is selected */
#define  SB_COLL        0x10   /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

* The low-water mark is always presumed to be 0 in the current implementation.

6.1.3. Socket connection queuing

In dealing with connection oriented sockets (e.g. SOCK_STREAM) the two ends are considered distinct. One end is termed *active*, and generates connection requests. The other end is called *passive* and accepts connection requests.

From the passive side, a socket is marked with SO_ACCEPTCONN when a *listen* call is made, creating two queues of sockets: *so_q0* for connections in progress and *so_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so_q0* by calling the routine *sonewconn*(). When the connection is established, the socket structure is then transferred to *so_q*, making it available for an *accept*.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped, with notification to the peers as appropriate.

6.2. Protocol layer(s)

Each socket is created in a communications domain, which usually implies both an addressing structure (address family) and a set of protocols which implement various socket types within the domain (protocol family). Each domain is defined by the following structure:

```
struct  domain {
        int     dom_family;            /* PF_xxx */
        char    *dom_name;
        int     (*dom_init)();         /* initialize domain data structures */
        int     (*dom_externalize)();  /* externalize access rights */
        int     (*dom_dispose)();      /* dispose of internalized rights */
        struct  protosw *dom_protosw, *dom_protoswNPROTOSW;
        struct  domain *dom_next;
};
```

At boot time, each domain configured into the kernel is added to a linked list of domain. The initialization procedure of each domain is then called. After that time, the domain structure is used to locate protocols within the protocol family. It may also contain procedure references for externalization of access rights at the receiving socket and the disposal of access rights that are not received.

Protocols are described by a set of entry points and certain socket-visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short   pr_type;                /* socket type used for */
        struct  domain *pr_domain;      /* domain protocol a member of */
        short   pr_protocol;            /* protocol number */
        short   pr_flags;               /* socket visible attributes */
/* protocol-protocol hooks */
        int     (*pr_input)();          /* input to protocol (from below) */
        int     (*pr_output)();         /* output to protocol (from above) */
        int     (*pr_ctlinput)();       /* control input (from below) */
        int     (*pr_ctloutput)();      /* control output (from above) */
/* user-protocol hook */
        int     (*pr_usrreq)();         /* user request */
/* utility hooks */
        int     (*pr_init)();           /* initialization routine */
        int     (*pr_fasttimo)();       /* fast timeout (200ms) */
        int     (*pr_slowtimo)();       /* slow timeout (500ms) */
        int     (*pr_drain)();          /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any of the arguments to these entries and must either pass it onward or dispose of it. (On output, the lowest level reached must free buffers storing the arguments; on input, the highest level is responsible for freeing buffers.)

The *pr_usrreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define PR_ATOMIC          0x01    /* exchange atomic messages only */
#define PR_ADDR            0x02    /* addresses given with messages */
#define PR_CONNREQUIRED    0x04    /* connection required by protocol */
#define PR_WANTRCVD        0x08    /* want PRU_RCVD calls */
#define PR_RIGHTS          0x10    /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notify the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates that any data placed in the socket's receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies that each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates that the protocol supports the passing of capabilities; this is currently used only by the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table for the domain looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (e.g. SOCK_STREAM), while the *pr_domain* is a back pointer to the domain structure. The *pr_protocol* field contains the protocol number of the protocol, normally a well-known value.

6.3. Network-interface layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and decapsulation of any link-layer header information required to deliver a message to its destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. An interface may have addresses in one or more address families. The address is set at boot time using an *ioctl* on a socket in the appropriate domain; this operation is implemented by the protocol family, after verifying the operation through the device *ioctl* entry.

An interface is defined by the following structure,

```
struct ifnet {
        char    *if_name;               /* name, e.g. "en" or "lo" */
        short   if_unit;                /* sub-unit for lower level driver */
        short   if_mtu;                 /* maximum transmission unit */
        short   if_flags;               /* up/down, broadcast, etc. */
        short   if_timer;               /* time 'til if_watchdog called */
        struct  ifaddr *if_addrlist;    /* list of addresses of interface */
        struct  ifqueue if_snd;         /* output queue */
        int     (*if_init)();           /* init routine */
        int     (*if_output)();         /* output routine */
        int     (*if_ioctl)();          /* ioctl routine */
        int     (*if_reset)();          /* bus reset routine */
        int     (*if_watchdog)();       /* timer routine */
        int     if_ipackets;            /* packets received on interface */
        int     if_ierrors;             /* input errors on interface */
        int     if_opackets;            /* packets sent on interface */
        int     if_oerrors;             /* output errors on interface */
        int     if_collisions;          /* collisions on csma interfaces */
        struct  ifnet *if_next;
};
```

Each interface address has the following form:

```
struct ifaddr {
        struct  sockaddr ifa_addr;      /* address of interface */
        union {
                struct  sockaddr ifu_broadaddr;
                struct  sockaddr ifu_dstaddr;
        } ifa_ifu;
        struct  ifnet *ifa_ifp;         /* back-pointer to interface */
        struct  ifaddr *ifa_next;       /* next address for interface */
};
#define ifa_broadaddr   ifa_ifu.ifu_broadaddr   /* broadcast address */
#define ifa_dstaddr     ifa_ifu.ifu_dstaddr     /* other end of p-to-p link */
```

The protocol generally maintains this structure as part of a larger structure containing additional information concerning the address.

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*; if *if_timer* is non-zero, it is decremented once per second until it reaches zero, at which time the watchdog routine is called.

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
#define  IFF_UP             0x1    /* interface is up */
#define  IFF_BROADCAST      0x2    /* broadcast is possible */
#define  IFF_DEBUG          0x4    /* turn on debugging */
#define  IFF_LOOPBACK       0x8    /* is a loopback net */
#define  IFF_POINTOPOINT    0x10   /* interface is point-to-point link */
#define  IFF_NOTRAILERS     0x20   /* avoid use of trailers */
#define  IFF_RUNNING        0x40   /* resources allocated */
#define  IFF_NOARP          0x80   /* no address resolution protocol */
```

If the interface is connected to a network which supports transmission of *broadcast* packets, the IFF_BROADCAST flag will be set and the *ifa_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point-to-point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *ifa_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets, or (where per-host negotiation of trailers is possible) that trailer encapsulations should not be requested; *trailer* protocols are described in section 14. The IFF_NOARP flag indicates the interface should not use an "address resolution protocol" in mapping internetwork addresses to local network addresses.

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS *ioctl*s. SIOCSIFADDR is used initially to define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration. The destination address of a point-to-point link is set with SIOCSIFDSTADDR. Corresponding operations exist to read each value. Protocol families may also support operations to set and read the broadcast address. In addition, the SIOCGIFCONF *ioctl* retrieves a list of interface names and addresses for all interfaces and protocols on the host.

6.3.1. UNIBUS interfaces

All hardware related interfaces currently reside on the UNIBUS. Consequently a common set of utility routines for dealing with the UNIBUS has been developed. Each UNIBUS interface utilizes a structure of the following form:

```
struct  ifubinfo {
        short     iff_uban;                  /* uba number */
        short     iff_hlen;                  /* local net header length */
        struct    uba_regs *iff_uba;         /* uba regs, in vm */
        short     iff_flags;                 /* used during uballoc's */
};
```

Additional structures are associated with each receive and transmit buffer, normally one each per interface; for read,

```
struct  ifrw {
        caddr_t   ifrw_addr;                        /* virt addr of header */
        short     ifrw_bdp;                         /* unibus bdp */
        short     ifrw_flags;                       /* type, etc. */
#define IFRW_W    0x01                              /* is a transmit buffer */
        int       ifrw_info;                        /* value from ubaalloc */
        int       ifrw_proto;                       /* map register prototype */
        struct    pte *ifrw_mr;                     /* base of map registers */
};
```

and for write,

```
struct  ifxmt {
        struct    ifrw ifrw;
        caddr_t   ifw_base;                         /* virt addr of buffer */
        struct    pte ifw_wmap[IF_MAXNUBAMR];       /* base pages for output */
        struct    mbuf *ifw_xtofree;                /* pages being dma'd out */
        short     ifw_xswapd;                       /* mask of clusters swapped */
        short     ifw_nmr;                          /* number of entries in wmap */
};
#define ifw_addr  ifrw.ifrw_addr
#define ifw_bdp   ifrw.ifrw_bdp
#define ifw_flags ifrw.ifrw_flags
#define ifw_info  ifrw.ifrw_info
#define ifw_proto ifrw.ifrw_proto
#define ifw_mr    ifrw.ifrw_mr
```

One of each of these structures is conveniently packaged for interfaces with single buffers for each direction, as follows:

```
struct  ifuba {
        struct    ifubinfo ifu_info;
        struct    ifrw ifu_r;
        struct    ifxmt ifu_xmt;
};
#define ifu_uban     ifu_info.iff_uban
#define ifu_hlen     ifu_info.iff_hlen
#define ifu_uba      ifu_info.iff_uba
#define ifu_flags    ifu_info.iff_flags
#define ifu_w        ifu_xmt.ifrw
#define ifu_xtofree  ifu_xmt.ifw_xtofree
```

The *if_ubinfo* structure contains the general information needed to characterize the I/O-mapped buffers for the device. In addition, there is a structure describing each buffer, including UNIBUS resources held by the interface. Sufficient memory pages and bus map registers are allocated to each buffer upon initialization according to the maximum packet size and header length. The kernel virtual address of the buffer is held in *ifrw_addr*, and the map registers begin at *ifrw_mr*. UNIBUS map register *ifrw_mr*[−1] maps the local network header ending on a page boundary. UNIBUS data paths are reserved for read and for write, given by *ifrw_bdp*. The prototype of the map registers for read and for write is saved in *ifrw_proto*.

When write transfers are not at least half-full pages on page boundaries, the data are just copied into the pages mapped on the UNIBUS and the transfer is started. If a write transfer is at least half a page long and on a page boundary, UNIBUS page table entries are swapped to reference the pages, and then the initial pages are remapped from *ifw_wmap* when the transfer completes. The mbufs containing the mapped pages are placed on the *ifw_xtofree* queue to be freed after transmission.

When read transfers give at least half a page of data to be input, page frames are allocated from a network page list and traded with the pages already containing the data, mapping the allocated pages to replace the input pages for the next UNIBUS data input.

The following utility routines are available for use in writing network interface drivers; all use the structures described above.

if_ubaminit(ifubinfo, uban, hlen, nmr, ifr, nr, ifx, nx);
if_ubainit(ifuba, uban, hlen, nmr);

> *if_ubaminit* allocates resources on UNIBUS adapter *uban*, storing the information in the *ifubinfo*, *ifrw* and *ifxmt* structures referenced. The *ifr* and *ifx* parameters are pointers to arrays of *ifrw* and *ifxmt* structures whose dimensions are *nr* and *nx*, respectively. *if_ubainit* is a simpler, backwards-compatible interface used for hardware with single buffers of each type. They are called only at boot time or after a UNIBUS reset. One data path (buffered or unbuffered, depending on the *ifu_flags* field) is allocated for each buffer. The *nmr* parameter indicates the number of UNIBUS mapping registers required to map a maximal sized packet onto the UNIBUS, while *hlen* specifies the size of a local network header, if any, which should be mapped separately from the data (see the description of trailer protocols in chapter 14). Sufficient UNIBUS mapping registers and pages of memory are allocated to initialize the input data path for an initial read. For the output data path, mapping registers and pages of memory are also allocated and mapped onto the UNIBUS. The pages associated with the output data path are held in reserve in the event a write requires copying non-page-aligned data (see *if_wubaput* below). If *if_ubainit* is called with memory pages already allocated, they will be used instead of allocating new ones (this normally occurs after a UNIBUS reset). A 1 is returned when allocation and initialization are successful, 0 otherwise.

m = if_ubaget(ifubinfo, ifr, totlen, off0, ifp);
m = if_rubaget(ifuba, totlen, off0, ifp);

> *if_ubaget* and *if_rubaget* pull input data out of an interface receive buffer and into an mbuf chain. The first interface passes pointers to the *ifubinfo* structure for the interface and the *ifrw* structure for the receive buffer; the second call may be used for single-buffered devices. *totlen* specifies the length of data to be obtained, not counting the local network header. If *off0* is non-zero, it indicates a byte offset to a trailing local network header which should be copied into a separate mbuf and prepended to the front of the resultant mbuf chain. When the data amount to at least a half a page, the previously mapped data pages are remapped into the mbufs and swapped with fresh pages, thus avoiding any copy. The receiving interface is recorded as *ifp*, a pointer to an *ifnet* structure, for the use of the receiving network protocol. A 0 return value indicates a failure to allocate resources.

if_wubaput(ifubinfo, ifx, m);
if_wubaput(ifuba, m);

> *if_ubaput* and *if_wubaput* map a chain of mbufs onto a network interface in preparation for output. The first interface is used by devices with multiple transmit buffers. The chain includes any local network header, which is copied so that it resides in the mapped and aligned I/O space. Page-aligned data that are page-aligned in the output buffer are mapped to the UNIBUS in place of the normal buffer page, and the corresponding mbuf is placed on a queue to be freed after transmission. Any other mbufs which contained non-page-sized data portions are copied to the I/O space and then freed. Pages mapped from a previous output operation (no longer needed) are unmapped.

7. Socket/protocol interface

The interface between the socket routines and the communication protocols is through the *pr_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define  PRU_ATTACH       0      /* attach protocol */
#define  PRU_DETACH       1      /* detach protocol */
#define  PRU_BIND         2      /* bind socket to address */
#define  PRU_LISTEN       3      /* listen for connection */
#define  PRU_CONNECT      4      /* establish connection to peer */
#define  PRU_ACCEPT       5      /* accept connection from peer */
#define  PRU_DISCONNECT   6      /* disconnect from peer */
#define  PRU_SHUTDOWN     7      /* won't send any more data */
#define  PRU_RCVD         8      /* have taken data; more room now */
#define  PRU_SEND         9      /* send this data */
#define  PRU_ABORT        10     /* abort (fast DISCONNECT, DETATCH) */
#define  PRU_CONTROL      11     /* control operations on protocol */
#define  PRU_SENSE        12     /* return status into m */
#define  PRU_RCVOOB       13     /* retrieve out of band data */
#define  PRU_SENDOOB      14     /* send out of band data */
#define  PRU_SOCKADDR     15     /* fetch socket's address */
#define  PRU_PEERADDR     16     /* fetch peer's address */
#define  PRU_CONNECT2     17     /* connect two sockets */
/* begin for protocols internal use */
#define  PRU_FASTTIMO     18     /* 200ms timeout */
#define  PRU_SLOWTIMO     19     /* 500ms timeout */
#define  PRU_PROTORCV     20     /* receive from below */
#define  PRU_PROTOSEND    21     /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(so, req, m, addr, rights);
int error; struct socket *so; int req; struct mbuf *m, *addr, *rights;
```

The mbuf data chain *m* is supplied for output operations and for certain other operations where it is to receive a result. The address *addr* is supplied for address-oriented requests such as PRU_BIND and PRU_CONNECT. The *rights* parameter is an optional pointer to an mbuf chain containing user-specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of the data mbuf chains on output operations. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH
: When a protocol is bound to a socket (with the *socket* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH
: This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND
: When a socket is initially created it has no address bound to it. This request indicates that an address should be bound to an existing socket. The protocol module must verify that the requested address is valid and available for use.

PRU_LISTEN
: The "listen" request indicates the user wishes to listen for incoming connection requests on

the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to accept a connection.

PRU_CONNECT
: The "connect" request indicates the user wants to a establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel81b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel80], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting,* it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT
: Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT
: Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN
: This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown and/or notify a connected peer of the shutdown.

PRU_RCVD
: This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND
: Each user request to send data is translated into one or more PRU_SEND requests (a protocol may indicate that a single user send request must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT
: This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL
: The "control" request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention). The *rights* parameter contains a pointer to an *ifnet* structure if the *ioctl* operation pertains to a particular network interface.

PRU_SENSE
: The "sense" request is generated when the user makes an *fstat* system call on a socket; it

requests status of the associated socket. This currently returns a standard *stat* structure. It typically contains only the optimal transfer size for the connection (based on buffer size, windowing information and maximum packet size). The *m* parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB
: Any "out-of-band" data presently available is to be returned. An mbuf is passed to the protocol module, and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf. An error may be returned if out-of-band data is not (yet) available or has already been consumed. The *addr* parameter contains any options such as MSG_PEEK to examine data without consuming it.

PRU_SENDOOB
: Like PRU_SEND, but for out-of-band data.

PRU_SOCKADDR
: The local address of the socket is returned, if any is currently bound to it. The address (with protocol specific format) is returned in the *addr* parameter.

PRU_PEERADDR
: The address of the peer to which the socket is connected is returned. The socket must be in a SS_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2
: The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol's operation (e.g. PRU_SLOWTIMO).

PRU_FASTTIMO
: A "fast timeout" has occurred. This request is made when a timeout occurs in the protocol's *pr_fastimo* routine. The *addr* parameter indicates which timer expired.

PRU_SLOWTIMO
: A "slow timeout" has occurred. This request is made when a timeout occurs in the protocol's *pr_slowtimo* routine. The *addr* parameter indicates which timer expired.

PRU_PROTORCV
: This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND
: This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked "addressed to protocol" instead of "addressed to user" are left to the protocol modules. No protocols currently use this facility.

8. Protocol/protocol interface

The interface between protocol modules is through the *pr_usrreq*, *pr_input*, *pr_output*, *pr_ctlinput*, and *pr_ctloutput* routines. The calling conventions for all but the *pr_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

8.1. pr_output

The Internet protocol UDP uses the convention,

```
error = udp_output(inp, m);
int error; struct inpcb *inp; struct mbuf *m;
```

where the *inp*, "*in*ternet *p*rotocol *c*ontrol *b*lock", passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on. UDP is based on the Internet Protocol, IP [Postel81a], as its transport. UDP passes a packet to the IP module for output as follows:

```
error = ip_output(m, opt, ro, flags);
int error; struct mbuf *m, *opt; struct route *ro; int flags;
```

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is is used in making routing decisions (and passing them back to the caller for use in subsequent calls). The final parameter, *flags* contains flags indicating whether the user is allowed to transmit a broadcast packet and if routing is to be performed. The broadcast flag may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occurred which could be detected immediately (no buffer space available, no route to destination, etc.).

8.2. pr_input

Both UDP and TCP use the following calling convention,

```
(void) (*protosw[].pr_input)(m, ifp);
struct mbuf *m; struct ifnet *ifp;
```

Each mbuf list passed is a single packet to be processed by the protocol module. The interface from which the packet was received is passed as the second parameter.

The IP input routine is a VAX software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission. The software interrupt is enabled by the network interfaces when they place input data on the input queue.

8.3. pr_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data).

The common calling convention for this routine is,

```
(void) (*protosw[].pr_ctlinput)(req, addr);
int req; struct sockaddr *addr;
```

The *req* parameter is one of the following,

```
#define PRC_IFDOWN                0    /* interface transition */
#define PRC_ROUTEDEAD             1    /* select new route if possible */
#define PRC_QUENCH                4    /* some said to slow down */
#define PRC_MSGSIZE               5    /* message size forced drop */
#define PRC_HOSTDEAD              6    /* normally from IMP */
#define PRC_HOSTUNREACH           7    /* ditto */
#define PRC_UNREACH_NET           8    /* no route to network */
#define PRC_UNREACH_HOST          9    /* no route to host */
#define PRC_UNREACH_PROTOCOL      10   /* dst says bad protocol */
#define PRC_UNREACH_PORT          11   /* bad port # */
#define PRC_UNREACH_NEEDFRAG      12   /* IP_DF caused drop */
#define PRC_UNREACH_SRCFAIL       13   /* source route failed */
#define PRC_REDIRECT_NET          14   /* net routing redirect */
#define PRC_REDIRECT_HOST         15   /* host routing redirect */
#define PRC_REDIRECT_TOSNET       14   /* redirect for type of service & net */
#define PRC_REDIRECT_TOSHOST      15   /* redirect for tos & host */
#define PRC_TIMXCEED_INTRANS      18   /* packet lifetime expired in transit */
#define PRC_TIMXCEED_REASS        19   /* lifetime expired on reass q */
#define PRC_PARAMPROB             20   /* header incorrect */
```

while the *addr* parameter is the address to which the condition applies. Many of the requests have obviously been derived from ICMP (the Internet Control Message Protocol [Postel81c]), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

8.4. pr_ctloutput

This is the routine that implements per-socket options at the protocol level for *getsockopt* and *setsockopt*. The calling convention is,

```
error = (*protosw[].pr_ctloutput)(op, so, level, optname, mp);
int op; struct socket *so; int level, optname; struct mbuf **mp;
```

where *op* is one of PRCO_SETOPT or PRCO_GETOPT, *so* is the socket from whence the call originated, and *level* and *optname* are the protocol level and option name supplied by the user. The results of a PRCO_GETOPT call are returned in an mbuf whose address is placed in *mp* before return. On a PRCO_SETOPT call, *mp* contains the address of an mbuf containing the option data; the mbuf should be freed before return.

9. Protocol/network-interface interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases with which to be concerned, transmission of a packet and receipt of a packet; each will be considered separately.

9.1. Packet transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable media, such as the Ethernet, "successful" transmission simply means that the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are only those that can be detected immediately, and are normally trivial in nature (no buffer space, address format not handled, etc.). No indication is received if errors are detected after the call has returned.

9.2. Packet reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In this system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued for the protocol module, and a VAX software interrupt is posted to initiate processing.

Three macros are available for queuing and dequeuing packets:

IF_ENQUEUE(ifq, m)
: This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)
: This places a pointer to the packet at the head of queue *ifq* in *m* and removes the packet from the queue. A zero value will be returned in *m* if the queue is empty.

IF_DEQUEUEIF(ifq, m, ifp)
: Like IF_DEQUEUE, this removes the next packet from the head of a queue and returns it in *m*. A pointer to the interface on which the packet was received is placed in *ifp*, a (struct ifnet *).

IF_PREPEND(ifq, m)
: This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to increment the count of the number of packets dropped, and the offending packet is dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

10. Gateways and routing issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

10.1. Routing tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long   rt_hash;                  /* hash key for lookups */
        struct   sockaddr rt_dst;          /* destination net or host */
        struct   sockaddr rt_gateway;      /* forwarding agent */
        short    rt_flags;                 /* see below */
        short    rt_refcnt;                /* no. of references to structure */
        u_long   rt_use;                   /* packets sent using route */
        struct   ifnet *rt_ifp;            /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). The first appropriate route discovered is used. By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to a "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (the desired final destination), a gateway to which to send the packet, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept, along with a count of "held references" to the dynamically allocated structure to insure that memory reclamation occurs only when the route is not in use. Finally, a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the *rt_dst* field during lookup. If the route is to a network, only a packet's destination network is compared to the *rt_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will contain the address of the eventual destination, while the local network header will address the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates that the route is to an "indirect" gateway agent, and that the local network header should be filled in from the *rt_gateway* field instead of from the final internetwork destination address.

It is assumed that multiple routes to the same destination will not be present; only one of multiple routes, that most recently installed, will be used.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Current connections may be rerouted after notification of the protocols by means of their *pr_ctlinput* entries. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

10.2. Routing table interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure containing the desired destination:

```
struct route {
        struct      rtentry *ro_rt;
        struct      sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until released with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, allocate and free routes whenever their destination address changes.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address, the new gateway to that destination, and the source of the redirect. Redirects are accepted only from the current router for the destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accessible from the host are ignored.

10.3. User level routing policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies that routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

Several routing policy processes have already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up-to-date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet EGP (Exterior Gateway Protocol), has been accomplished using a similar process.

11. Raw sockets

A raw socket is an object which allows users direct access to a lower-level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

11.1. Control blocks

Every raw socket has a protocol control block of the following form:

```
struct rawcb {
        struct   rawcb *rcb_next;        /* doubly linked list */
        struct   rawcb *rcb_prev;
        struct   socket *rcb_socket;     /* back pointer to socket */
        struct   sockaddr rcb_faddr;     /* destination address */
        struct   sockaddr rcb_laddr;     /* socket's address */
        struct   sockproto rcb_proto;    /* protocol family, protocol */
        caddr_t  rcb_pcb;                /* protocol specific stuff */
        struct   mbuf *rcb_options;      /* protocol specific options */
        struct   route rcb_route;        /* routing information */
        short    rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The *rcb_proto* structure contains the protocol family and protocol number with which the raw socket is associated. The protocol, family and addresses are used to filter packets on input; this will be described in more detail shortly. If any protocol-specific information is required, it may be attached to the control block using the *rcb_pcb* field. Protocol-specific options for transmission in outgoing packets may be stored in *rcb_options*.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Routing is expected to be performed by the underlying protocol if necessary.

11.2. Input processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives unassigned packets to the raw input routine with the call:

```
        raw_input(m, proto, src, dst)
        struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
        struct raw_header {
                struct      sockproto raw_proto;
                struct      sockaddr raw_dst;
                struct      sockaddr raw_src;
        };
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1) The protocol family of the socket and header agree.
2) If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.
3) If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.
4) The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be "block compared".

11.3. Output processing

On output the raw *pr_usrreq* routine passes the packet and a pointer to the raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12. Buffering and congestion control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for "normal" network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be "turned off" as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

12.1. Memory management

The basic memory allocation routines manage a private page map, the size of which determines the maximum amount of memory that may be allocated by the network. A small amount of memory is allocated at boot time to initialize the mbuf and mbuf page cluster free lists. When the free lists are exhausted, more memory is requested from the system memory allocator if space remains in the map. If memory cannot be allocated, callers may block awaiting free memory, or the failure may be reflected to the caller immediately. The allocator will not block awaiting free map entries, however, as exhaustion of the page map usually indicates that buffers have been lost due to a "leak." The private page table is used by the network buffer management routines in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple references to a page are present.

Mbufs are 128 byte structures, 8 fitting in a 1Kbyte page of memory. When data is placed in mbufs, it is copied or remapped into logically contiguous pages of memory from the network page pool if possible. Data smaller than half of the size of a page is copied into one or more 112 byte mbuf data areas.

12.2. Protocol buffering policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the "silly window syndrome" described in [Clark82] at both the sending and receiving ends.

12.3. Queue limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a "defensive" mechanism the queue limits may be adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

12.4. Packet forwarding

When packets can not be forwarded because of memory limitations, the system attempts to generate a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

13. Out of band data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocol's prerogative to support larger-sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and is usually not stored in the socket's receive queue. A socket-level option, SO_OOBINLINE, is provided to force out-of-band data to be placed in the normal receive queue when urgent data is received; this sometimes amelioriates problems due to loss of data when multiple out-of-band segments are received before the first has been passed to the user. The PRU_SENDOOB and PRU_RCVOOB requests to the *pr_usrreq* routine are used in sending and receiving data.

14. Trailer protocols

Core to core copies can be expensive. Consequently, a great deal of effort was spent in minimizing such operations. The VAX architecture provides virtual memory hardware organized in page units. To cut down on copy operations, data is kept in page-sized units on page-aligned boundaries whenever possible. This allows data to be moved in memory simply by remapping the page instead of copying. The mbuf and network interface routines perform page table manipulations where needed, hiding the complexities of the VAX virtual memory hardware from higher level code.

Data enters the system in two ways: from the user, or from the network (hardware interface). When data is copied from the user's address space into the system it is deposited in pages (if sufficient data is present). This encourages the user to transmit information in messages which are a multiple of the system page size.

Unfortunately, performing a similar operation when taking data from the network is very difficult. Consider the format of an incoming packet. A packet usually contains a local network header followed by one or more headers used by the high level protocols. Finally, the data, if any, follows these headers. Since the header information may be variable length, DMA'ing the eventual data for the user into a page aligned area of memory is impossible without *a priori* knowledge of the format (e.g., by supporting only a single protocol header format).

To allow variable length header information to be present and still ensure page alignment of data, a special local network encapsulation may be used. This encapsulation, termed a *trailer protocol* [Leffler84], places the variable length header information after the data. A fixed size local network header is then prepended to the resultant packet. The local network header contains the size of the data portion (in units of 512 bytes), and a new *trailer protocol header*, inserted before the variable length information, contains the size of the variable length header information. The following trailer protocol header is used to store information regarding the variable length protocol header:

```
struct {
        short       protocol;           /* original protocol no. */
        short       length;             /* length of trailer */
};
```

The processing of the trailer protocol is very simple. On output, the local network header indicates that a trailer encapsulation is being used. The header also includes an indication of the number of data pages present before the trailer protocol header. The trailer protocol header is initialized to contain the actual protocol identifier and the variable length header size, and is appended to the data along with the variable length header information.

On input, the interface routines identify the trailer encapsulation by the protocol type stored in the local network header, then calculate the number of pages of data to find the beginning of the trailer. The trailing information is copied into a separate mbuf and linked to the front of the resultant packet.

Clearly, trailer protocols require cooperation between source and destination. In addition, they are normally cost effective only when sizable packets are used. The current scheme works because the local network encapsulation header is a fixed size, allowing DMA operations to be performed at a known offset from the first data page being received. Should the local network header be variable length this scheme fails.

Statistics collected indicate that as much as 200Kb/s can be gained by using a trailer protocol with 1Kbyte packets. The average size of the variable length header was 40 bytes (the size of a minimal TCP/IP packet header). If hardware supports larger sized packets, even greater gains may be realized.

Acknowledgements

The internal structure of the system is patterned after the Xerox PUP architecture [Boggs79], while in certain places the Internet protocol family has had a great deal of influence in the design. The use of software interrupts for process invocation is based on similar facilities found in the VMS operating system. Many of the ideas related to protocol modularity, memory management, and network interfaces are based on Rob Gurwitz's TCP/IP implementation for the 4.1BSD version of UNIX on the VAX [Gurwitz81]. Greg Chesson explained his use of trailer encapsulations in Datakit, instigating their use in our system.

References

[Boggs79] Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture.* Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78] Bolt Beranek and Newman; Specification for the Interconnection of Host and IMP. BBN Technical Report 1822. May 1978.

[Cerf78] Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82] Clark, D. D.; Window and Acknowledgement Strategy in TCP, RFC-813. Network Information Center, SRI International. July 1982.

[DEC80] Digital Equipment Corporation; *DECnet DIGITAL Network Architecture – General Description.* Order No. AA-K179A-TK. October 1980.

[Gurwitz81] Gurwitz, R. F.; VAX-UNIX Networking Support Project – Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[ISO81] International Organization for Standardization. *ISO Open Systems Interconnection – Basic Reference Model.* ISO/TC 97/SC 16 N 719. August 1981.

[Joy86] Joy, W.; Fabry, R.; Leffler, S.; McKusick, M.; and Karels, M.; Berkeley Software Architecture Manual, 4.3BSD Edition. *UNIX Programmer's Supplementary Documents*, Vol. 1 (PS1:6). Computer Systems Research Group, University of California, Berkeley. May, 1986.

[Leffler84]	Leffler, S.J. and Karels, M.J.; Trailer Encapsulations, RFC-893. Network Information Center, SRI International. April 1984.
[Postel80]	Postel, J. User Datagram Protocol, RFC-768. Network Information Center, SRI International. May 1980.
[Postel81a]	Postel, J., ed. Internet Protocol, RFC-791. Network Information Center, SRI International. September 1981.
[Postel81b]	Postel, J., ed. Transmission Control Protocol, RFC-793. Network Information Center, SRI International. September 1981.
[Postel81c]	Postel, J. Internet Control Message Protocol, RFC-792. Network Information Center, SRI International. September 1981.
[Xerox81]	Xerox Corporation. *Internet Transport Protocols.* Xerox System Integration Standard 028112. December 1981.
[Zimmermann80]	Zimmermann, H. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications.* Com-28(4); 425-432. April 1980.

Appendix F – An Introductory Interprocess Communication Tutorial

The following appendix contains a document produced by Stuart Sechrest of the Computer Science Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science at the University of California, Berkeley, titled "*An Introductory 4.3BSD Interprocess Communication Tutorial*". The document describes in a simple way the use of pipes, socketpairs, sockets, and the use of datagram and stream communication. The intent of this document is to present a few simple example programs, not to describe the networking system in full.

An Introductory 4.3BSD Interprocess Communication Tutorial

Stuart Sechrest

Computer Science Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

ABSTRACT

Berkeley UNIX† 4.3BSD offers several choices for interprocess communication. To aid the programmer in developing programs which are comprised of cooperating processes, the different choices are discussed and a series of example programs are presented. These programs demonstrate in a simple way the use of pipes, socketpairs, sockets and the use of datagram and stream communication. The intent of this document is to present a few simple example programs, not to describe the networking system in full.

1. Goals

Facilities for interprocess communication (IPC) and networking were a major addition to UNIX in the Berkeley UNIX 4.2BSD release. These facilities required major additions and some changes to the system interface. The basic idea of this interface is to make IPC similar to file I/O. In UNIX a process has a set of I/O descriptors, from which one reads and to which one writes. Descriptors may refer to normal files, to devices (including terminals), or to communication channels. The use of a descriptor has three phases: its creation, its use for reading and writing, and its destruction. By using descriptors to write files, rather than simply naming the target file in the write call, one gains a surprising amount of flexibility. Often, the program that creates a descriptor will be different from the program that uses the descriptor. For example the shell can create a descriptor for the output of the 'ls' command that will cause the listing to appear in a file rather than on a terminal. Pipes are another form of descriptor that have been used in UNIX for some time. Pipes allow one-way data transmission from one process to another; the two processes and the pipe must be set up by a common ancestor.

The use of descriptors is not the only communication interface provided by UNIX. The signal mechanism sends a tiny amount of information from one process to another. The signaled process receives only the signal type, not the identity of the sender, and the number of possible signals is small. The signal semantics limit the flexibility of the signaling mechanism as a means of interprocess communication.

The identification of IPC with I/O is quite longstanding in UNIX and has proved quite successful. At first, however, IPC was limited to processes communicating within a single machine. With Berkeley UNIX 4.2BSD this expanded to include IPC between machines. This expansion has necessitated some change in the way that descriptors are created. Additionally, new possibilities for the meaning of read and write have been admitted. Originally the meanings, or semantics, of these terms were fairly simple. When you wrote something it was delivered. When you read something, you were blocked until the data arrived. Other possibilities exist, however. One can write without full assurance of delivery if one can check later to catch occasional failures. Messages can be kept as discrete units or merged into a stream. One can ask

† UNIX is a trademark of AT&T Bell Laboratories.

to read, but insist on not waiting if nothing is immediately available. These new possibilities are allowed in the Berkeley UNIX IPC interface.

Thus Berkeley UNIX 4.3BSD offers several choices for IPC. This paper presents simple examples that illustrate some of the choices. The reader is presumed to be familiar with the C programming language [Kernighan & Ritchie 1978], but not necessarily with the system calls of the UNIX system or with processes and interprocess communication. The paper reviews the notion of a process and the types of communication that are supported by Berkeley UNIX 4.3BSD. A series of examples are presented that create processes that communicate with one another. The programs show different ways of establishing channels of communication. Finally, the calls that actually transfer data are reviewed. To clearly present how communication can take place, the example programs have been cleared of anything that might be construed as useful work. They can, therefore, serve as models for the programmer trying to construct programs which are comprised of cooperating processes.

2. Processes

A *program* is both a sequence of statements and a rough way of referring to the computation that occurs when the compiled statements are run. A *process* can be thought of as a single line of control in a program. Most programs execute some statements, go through a few loops, branch in various directions and then end. These are single process programs. Programs can also have a point where control splits into two independent lines, an action called *forking*. In UNIX these lines can never join again. A call to the system routine *fork()*, causes a process to split in this way. The result of this call is that two independent processes will be running, executing exactly the same code. Memory values will be the same for all values set before the fork, but, subsequently, each version will be able to change only the value of its own copy of each variable. Initially, the only difference between the two will be the value returned by *fork()*. The parent will receive a process id for the child, the child will receive a zero. Calls to *fork()*, therefore, typically precede, or are included in, an if-statement.

A process views the rest of the system through a private table of descriptors. The descriptors can represent open files or sockets (sockets are communication objects that will be discussed below). Descriptors are referred to by their index numbers in the table. The first three descriptors are often known by special names, *stdin, stdout* and *stderr*. These are the standard input, output and error. When a process forks, its descriptor table is copied to the child. Thus, if the parent's standard input is being taken from a terminal (devices are also treated as files in UNIX), the child's input will be taken from the same terminal. Whoever reads first will get the input. If, before forking, the parent changes its standard input so that it is reading from a new file, the child will take its input from the new file. It is also possible to take input from a socket, rather than from a file.

3. Pipes

Most users of UNIX know that they can pipe the output of a program "prog1" to the input of another, "prog2," by typing the command *"prog1 | prog2."* This is called "piping" the output of one program to another because the mechanism used to transfer the output is called a pipe. When the user types a command, the command is read by the shell, which decides how to execute it. If the command is simple, for example, *"prog1,"* the shell forks a process, which executes the program, prog1, and then dies. The shell waits for this termination and then prompts for the next command. If the command is a compound command, *"prog1 | prog2,"* the shell creates two processes connected by a pipe. One process runs the program, prog1, the other runs prog2. The pipe is an I/O mechanism with two ends, or sockets. Data that is written into one socket can be read from the other.

Since a program specifies its input and output only by the descriptor table indices, which appear as variables or constants, the input source and output destination can be changed without changing the text of the program. It is in this way that the shell is able to set up pipes. Before executing prog1, the process can close whatever is at *stdout* and replace it with one end of a pipe. Similarly, the process that will execute prog2 can substitute the opposite end of the pipe for *stdin*.

```
#include <stdio.h>

#define DATA "Bright star, would I were steadfast as thou art . . ."

/*
 * This program creates a pipe, then forks.  The child communicates to the
 * parent over the pipe. Notice that a pipe is a one-way communications
 * device.  I can write to the output socket (sockets[1], the second socket
 * of the array returned by pipe()) and read from the input socket
 * (sockets[0]), but not vice versa.
 */

main()
{
      int sockets[2], child;

      /* Create a pipe */
      if (pipe(sockets) < 0) {
            perror("opening stream socket pair");
            exit(10);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) {
            char buf[1024];

            /* This is still the parent.  It reads the child's message. */
            close(sockets[1]);
            if (read(sockets[0], buf, 1024) < 0)
                  perror("reading message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      } else {
            /* This is the child.  It writes a message to its parent. */
            close(sockets[0]);
            if (write(sockets[1], DATA, sizeof(DATA)) < 0)
                  perror("writing message");
            close(sockets[1]);
      }
}
```

Figure 1 Use of a pipe

Let us now examine a program that creates a pipe for communication between its child and itself (Figure 1). A pipe is created by a parent process, which then forks. When a process forks, the parent's descriptor table is copied into the child's.

In Figure 1, the parent process makes a call to the system routine *pipe()*. This routine creates a pipe and places descriptors for the sockets for the two ends of the pipe in the process's descriptor table. *Pipe()* is passed an array into which it places the index numbers of the sockets it created. The two ends are not equivalent. The socket whose index is returned in the low word of the array is opened for reading only, while the socket in the high end is opened only for writing. This corresponds to the fact that the standard input is the first descriptor of a process's descriptor table and the standard output is the second. After creating the pipe, the parent creates the child with which it will share the pipe by calling *fork()*. Figure 2 illustrates the effect of a fork. The parent process's descriptor table points to both ends of the pipe. After the fork, both parent's and child's descriptor tables point to the pipe. The child can then use the pipe to send a message to the parent.

Just what is a pipe? It is a one-way communication mechanism, with one end opened for reading and the other end for writing. Therefore, parent and child need to agree on which way to turn the pipe, from parent to child or the other way around. Using the same pipe for communication both from parent to child and from child to parent would be possible (since both processes have references to both ends), but very complicated. If the parent and child are to have a two-way conversation, the parent creates two pipes, one for use in each direction. (In accordance with their plans, both parent and child in the example above close the socket that they will not use. It is not required that unused descriptors be closed, but it is good

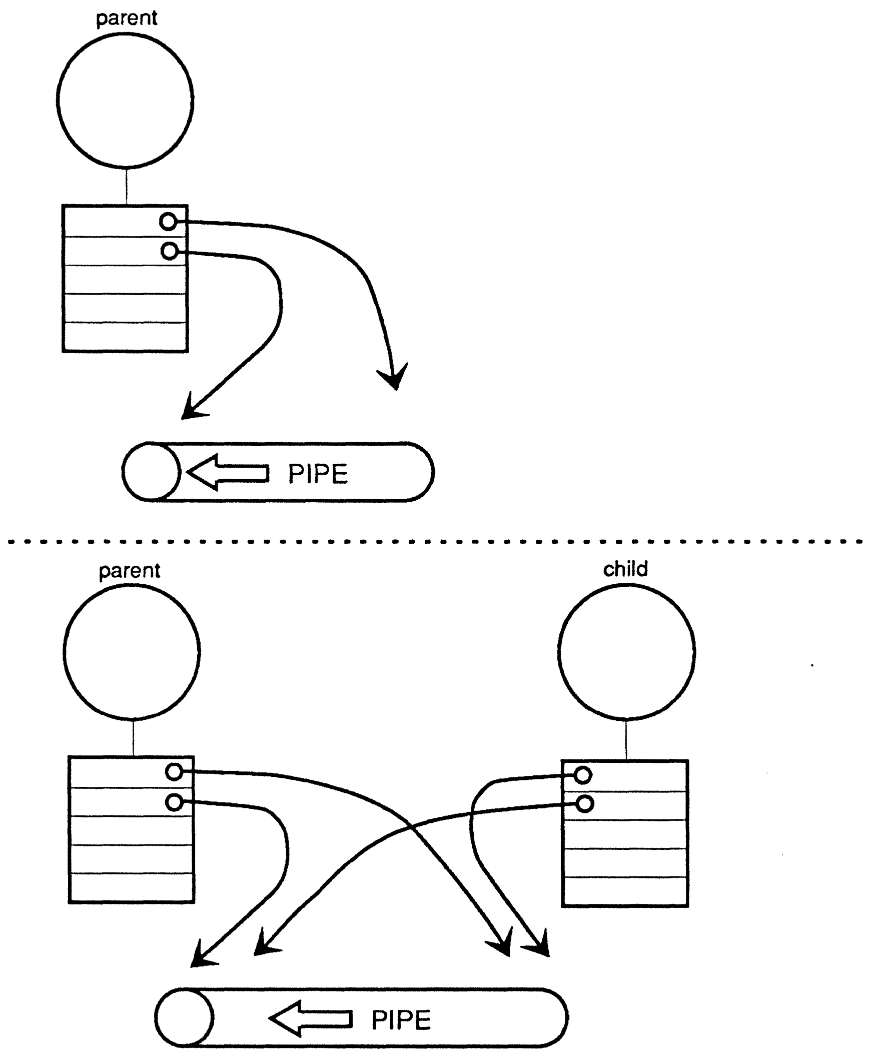

Figure 2 Sharing a pipe between parent and child

practice.) A pipe is also a *stream* communication mechanism; that is, all messages sent through the pipe are placed in order and reliably delivered. When the reader asks for a certain number of bytes from this stream, he is given as many bytes as are available, up to the amount of the request. Note that these bytes may have come from the same call to *write()* or from several calls to *write()* which were concatenated.

4. Socketpairs

Berkeley UNIX 4.3BSD provides a slight generalization of pipes. A pipe is a pair of connected sockets for one-way stream communication. One may obtain a pair of connected sockets for two-way stream communication by calling the routine *socketpair()*. The program in Figure 3 calls *socketpair()* to create such a connection. The program uses the link for communication in both directions. Since socketpairs are an extension of pipes, their use resembles that of pipes. Figure 4 illustrates the result of a fork following a call to *socketpair()*.

Socketpair() takes as arguments a specification of a domain, a style of communication, and a protocol. These are the parameters shown in the example. Domains and protocols will be discussed in the next section. Briefly, a domain is a space of names that may be bound to sockets and implies certain other conventions. Currently, socketpairs have only been implemented for one domain, called the UNIX domain. The UNIX domain uses UNIX path names for naming sockets. It only allows communication between sockets on the same machine.

Note that the header files <*sys/socket.h*> and <*sys/types.h*>. are required in this program. The constants AF_UNIX and SOCK_STREAM are defined in <*sys/socket.h*>, which in turn requires the file <*sys/types.h*> for some of its definitions.

5. Domains and Protocols

Pipes and socketpairs are a simple solution for communicating between a parent and child or between child processes. What if we wanted to have processes that have no common ancestor with whom to set up communication? Neither standard UNIX pipes nor socketpairs are the answer here, since both mechanisms require a common ancestor to set up the communication. We would like to have two processes separately create sockets and then have messages sent between them. This is often the case when providing or using a service in the system. This is also the case when the communicating processes are on separate machines. In Berkeley UNIX 4.3BSD one can create individual sockets, give them names and send messages between them.

Sockets created by different programs use names to refer to one another; names generally must be translated into addresses for use. The space from which an address is drawn is referred to as a *domain*. There are several domains for sockets. Two that will be used in the examples here are the UNIX domain (or AF_UNIX, for Address Format UNIX) and the Internet domain (or AF_INET). UNIX domain IPC is an experimental facility in 4.2BSD and 4.3BSD. In the UNIX domain, a socket is given a path name within the file system name space. A file system node is created for the socket and other processes may then refer to the socket by giving the proper pathname. UNIX domain names, therefore, allow communication between any two processes that work in the same file system. The Internet domain is the UNIX implementation of the DARPA Internet standard protocols IP/TCP/UDP. Addresses in the Internet domain consist of a machine network address and an identifying number, called a port. Internet domain names allow communication between machines.

Communication follows some particular ''style.'' Currently, communication is either through a *stream* or by *datagram*. Stream communication implies several things. Communication takes place across a connection between two sockets. The communication is reliable, error-free, and, as in pipes, no message boundaries are kept. Reading from a stream may result in reading the data sent from one or several calls to *write()* or only part of the data from a single call, if there is not enough room for the entire message, or if not all the data from a large message has been transferred. The protocol implementing such a style will retransmit messages received with errors. It will also return error messages if one tries to send a message after the connection has been broken. Datagram communication does not use connections. Each message is addressed individually. If the address is correct, it will generally be received, although this is not

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

#define DATA1 "In Xanadu, did Kublai Khan . . ."
#define DATA2 "A stately pleasure dome decree . . ."

/*
 * This program creates a pair of connected sockets then forks and
 * communicates over them.  This is very similar to communication with pipes,
 * however, socketpairs are two-way communications objects. Therefore I can
 * send messages in both directions.
 */

main()
{
      int sockets[2], child;
      char buf[1024];

      if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets) < 0) {
            perror("opening stream socket pair");
            exit(1);
      }

      if ((child = fork()) == -1)
            perror("fork");
      else if (child) { /* This is the parent. */
            close(sockets[0]);
            if (read(sockets[1], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            if (write(sockets[1], DATA2, sizeof(DATA2)) < 0)
                  perror("writing stream message");
            close(sockets[1]);
      } else {          /* This is the child. */
            close(sockets[1]);
            if (write(sockets[0], DATA1, sizeof(DATA1)) < 0)
                  perror("writing stream message");
            if (read(sockets[0], buf, 1024, 0) < 0)
                  perror("reading stream message");
            printf("-->%s\n", buf);
            close(sockets[0]);
      }
}
```

Figure 3 Use of a socketpair

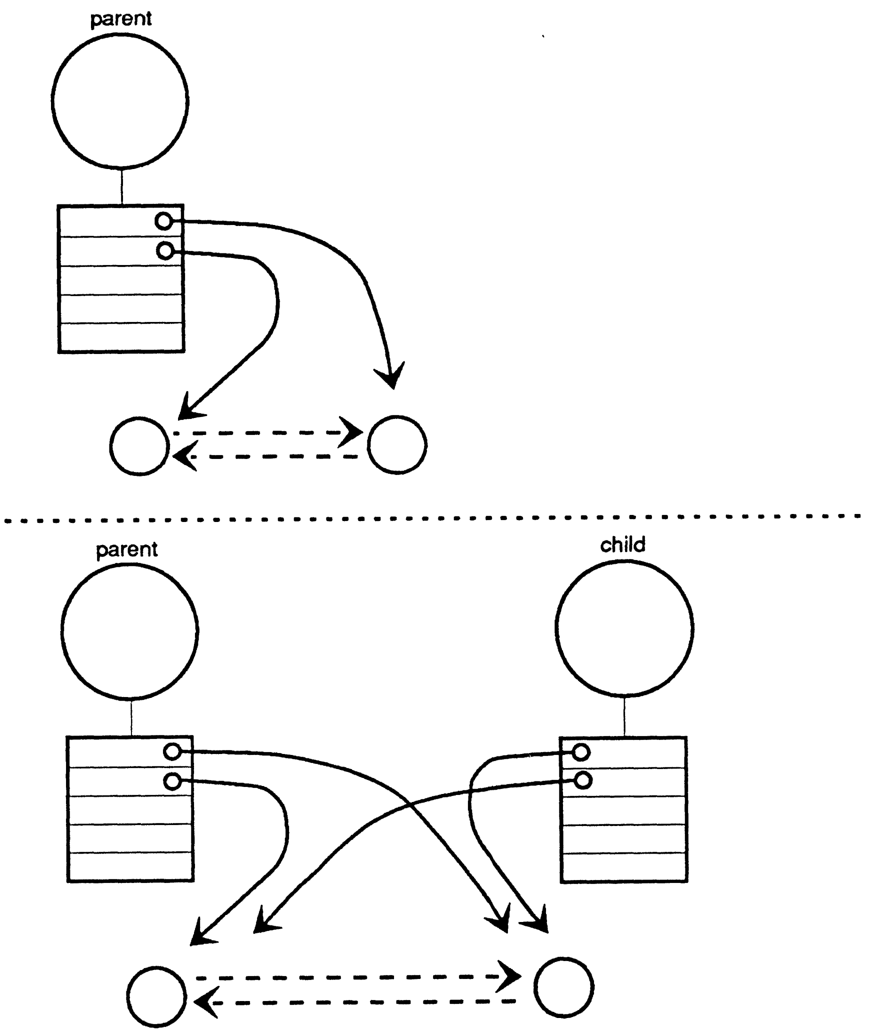

Figure 4 Sharing a socketpair between parent and child

guaranteed. Often datagrams are used for requests that require a response from the recipient. If no response arrives in a reasonable amount of time, the request is repeated. The individual datagrams will be kept separate when they are read, that is, message boundaries are preserved.

The difference in performance between the two styles of communication is generally less important than the difference in semantics. The performance gain that one might find in using datagrams must be weighed against the increased complexity of the program, which must now concern itself with lost or out of order messages. If lost messages may simply be ignored, the quantity of traffic may be a consideration. The expense of setting up a connection is best justified by frequent use of the connection. Since the performance of a protocol changes as it is tuned for different situations, it is best to seek the most up-to-date information when making choices for a program in which performance is crucial.

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

One specifies the domain, style and protocol of a socket when it is created. For example, in Figure 5a the call to *socket()* causes the creation of a datagram socket with the default protocol in the UNIX domain.

6. Datagrams in the UNIX Domain

Let us now look at two programs that create sockets separately. The programs in Figures 5a and 5b use datagram communication rather than a stream. The structure used to name UNIX domain sockets is defined in the file <*sys/un.h*>. The definition has also been included in the example for clarity.

Each program creates a socket with a call to *socket()*. These sockets are in the UNIX domain. Once a name has been decided upon it is attached to a socket by the system call *bind()*. The program in Figure 5a uses the name ''socket'', which it binds to its socket. This name will appear in the working directory of the program. The routines in Figure 5b use its socket only for sending messages. It does not create a name for the socket because no other process has to refer to it.

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

There is no established means of communicating names to interested parties. In the example, the program in Figure 5b gets the name of the socket to which it will send its message through its command line arguments. Once a line of communication has been created, one can send the names of additional, perhaps new, sockets over the link. Facilities will have to be built that will make the distribution of names less of a problem than it now is.

7. Datagrams in the Internet Domain

The examples in Figure 6a and 6b are very close to the previous example except that the socket is in the Internet domain. The structure of Internet domain addresses is defined in the file <*netinet/in.h*>. Internet addresses specify a host address (a 32-bit number) and a delivery slot, or port, on that machine. These ports are managed by the system routines that implement a particular protocol. Unlike UNIX domain names, Internet socket names are not entered into the file system and, therefore, they do not have to be unlinked after the socket has been closed. When a message must be sent between machines it is sent to the protocol routine on the destination machine, which interprets the address to determine to which socket the message should be delivered. Several different protocols may be active on the same machine, but, in general, they will not communicate with one another. As a result, different protocols are allowed to use the same port numbers. Thus, implicitly, an Internet address is a triple including a protocol as well as the port and machine address. An *association* is a temporary or permanent specification of a pair of communicating sockets. An association is thus identified by the tuple <*protocol, local machine address, local port, remote machine address, remote port*>. An association may be transient when using datagram sockets; the association actually exists during a *send* operation.

The protocol for a socket is chosen when the socket is created. The local machine address for a socket can be any valid network address of the machine, if it has more than one, or it can be the wildcard

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

/*
 * In the included file <sys/un.h> a sockaddr_un is defined as follows
 * struct sockaddr_un {
 *    short sun_family;
 *    char  sun_path[108];
 * };
 */

#include <stdio.h>

#define NAME "socket"

/*
 * This program creates a UNIX domain datagram socket, binds a name to it,
 * then reads from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_un name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, NAME);
      if (bind(sock, &name, sizeof(struct sockaddr_un))) {
            perror("binding name to datagram socket");
            exit(1);
      }
      printf("socket -->%s\n", NAME);
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
      unlink(NAME);
}
```

Figure 5a Reading UNIX domain datagrams

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

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is udgramsend pathname
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un name;

      /* Create socket on which to send. */
      sock = socket(AF_UNIX, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Construct name of socket to send to. */
      name.sun_family = AF_UNIX;
      strcpy(name.sun_path, argv[1]);
      /* Send message. */
      if (sendto(sock, DATA, sizeof(DATA), 0,
          &name, sizeof(struct sockaddr_un)) < 0) {
            perror("sending datagram message");
      }
      close(sock);
}
```

Figure 5b Sending a UNIX domain datagrams

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

/*
 * In the included file <netinet/in.h> a sockaddr_in is defined as follows:
 * struct sockaddr_in {
 *    short sin_family;
 *    u_short     sin_port;
 *    struct in_addr sin_addr;
 *    char  sin_zero[8];
 * };
 *
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */
main()
{
      int sock, length;
      struct sockaddr_in name;
      char buf[1024];

      /* Create socket from which to read. */
      sock = socket(AF_INET, SOCK_DGRAM, 0);
      if (sock < 0) {
            perror("opening datagram socket");
            exit(1);
      }
      /* Create name with wildcards. */
      name.sin_family = AF_INET;
      name.sin_addr.s_addr = INADDR_ANY;
      name.sin_port = 0;
      if (bind(sock, &name, sizeof(name))) {
            perror("binding datagram socket");
            exit(1);
      }
      /* Find assigned port value and print it out. */
      length = sizeof(name);
      if (getsockname(sock, &name, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(name.sin_port));
      /* Read from the socket */
      if (read(sock, buf, 1024) < 0)
            perror("receiving datagram packet");
      printf("-->%s\n", buf);
      close(sock);
}
```

Figure 6a Reading Internet domain datagrams

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

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is dgramsend hostname
 * portnumber
 */

main(argc, argv)
	int argc;
	char *argv[];
{
	int sock;
	struct sockaddr_in name;
	struct hostent *hp, *gethostbyname();

	/* Create socket on which to send. */
	sock = socket(AF_INET, SOCK_DGRAM, 0);
	if (sock < 0) {
		perror("opening datagram socket");
		exit(1);
	}
	/*
	 * Construct name, with no wildcards, of the socket to send to.
	 * Getnostbyname() returns a structure including the network address
	 * of the specified host.  The port number is taken from the command
	 * line.
	 */
	hp = gethostbyname(argv[1]);
	if (hp == 0) {
		fprintf(stderr, "%s: unknown host0, argv[1]);
		exit(2);
	}
	bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
	name.sin_family = AF_INET;
	name.sin_port = htons(atoi(argv[2]));
	/* Send message. */
	if (sendto(sock, DATA, sizeof(DATA), 0, &name, sizeof(name)) < 0)
		perror("sending datagram message");
	close(sock);
}
```

Figure 6b Sending an Internet domain datagram

value INADDR_ANY. The wildcard value is used in the program in Figure 6a. If a machine has several network addresses, it is likely that messages sent to any of the addresses should be deliverable to a socket. This will be the case if the wildcard value has been chosen. Note that even if the wildcard value is chosen, a program sending messages to the named socket must specify a valid network address. One can be willing to receive from "anywhere," but one cannot send a message "anywhere." The program in Figure 6b is given the destination host name as a command line argument. To determine a network address to which it can send the message, it looks up the host address by the call to *gethostbyname()*. The returned structure includes the host's network address, which is copied into the structure specifying the destination of the message.

The port number can be thought of as the number of a mailbox, into which the protocol places one's messages. Certain daemons, offering certain advertised services, have reserved or "well-known" port numbers. These fall in the range from 1 to 1023. Higher numbers are available to general users. Only servers need to ask for a particular number. The system will assign an unused port number when an address is bound to a socket. This may happen when an explicit *bind* call is made with a port number of 0, or when a *connect* or *send* is performed on an unbound socket. Note that port numbers are not automatically reported back to the user. After calling *bind()*, asking for port 0, one may call *getsockname()* to discover what port was actually assigned. The routine *getsockname()* will not work for names in the UNIX domain.

The format of the socket address is specified in part by standards within the Internet domain. The specification includes the order of the bytes in the address. Because machines differ in the internal representation they ordinarily use to represent integers, printing out the port number as returned by *getsockname()* may result in a misinterpretation. To print out the number, it is necessary to use the routine *ntohs()* (for *network to host: short*) to convert the number from the network representation to the host's representation. On some machines, such as 68000-based machines, this is a null operation. On others, such as VAXes, this results in a swapping of bytes. Another routine exists to convert a short integer from the host format to the network format, called *htons()*; similar routines exist for long integers. For further information, refer to the entry for *byteorder* in section 3 of the manual.

8. Connections

To send data between stream sockets (having communication style SOCK_STREAM), the sockets must be connected. Figures 7a and 7b show two programs that create such a connection. The program in 7a is relatively simple. To initiate a connection, this program simply creates a stream socket, then calls *connect()*, specifying the address of the socket to which it wishes its socket connected. Provided that the target socket exists and is prepared to handle a connection, connection will be complete, and the program can begin to send messages. Messages will be delivered in order without message boundaries, as with pipes. The connection is destroyed when either socket is closed (or soon thereafter). If a process persists in sending messages after the connection is closed, a SIGPIPE signal is sent to the process by the operating system. Unless explicit action is taken to handle the signal (see the manual page for *signal* or *sigvec*), the process will terminate and the shell will print the message "broken pipe."

Forming a connection is asymmetrical; one process, such as the program in Figure 7a, requests a connection with a particular socket, the other process accepts connection requests. Before a connection can be accepted a socket must be created and an address bound to it. This situation is illustrated in the top half of Figure 8. Process 2 has created a socket and bound a port number to it. Process 1 has created an unnamed socket. The address bound to process 2's socket is then made known to process 1 and, perhaps to several other potential communicants as well. If there are several possible communicants, this one socket might receive several requests for connections. As a result, a new socket is created for each connection. This new socket is the endpoint for communication within this process for this connection. A connection may be destroyed by closing the corresponding socket.

The program in Figure 7b is a rather trivial example of a server. It creates a socket to which it binds a name, which it then advertises. (In this case it prints out the socket number.) The program then calls *listen()* for this socket. Since several clients may attempt to connect more or less simultaneously, a queue of pending connections is maintained in the system address space. *Listen()* marks the socket as willing to accept connections and initializes the queue. When a connection is requested, it is listed in the queue. If the queue is full, an error status may be returned to the requester. The maximum length of this queue is specified by the second argument of *listen()*; the maximum length is limited by the system. Once the listen

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

#define DATA "Half a league, half a league . . ."

/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line.  One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the command
 * line is streamwrite hostname portnumber
 */

main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_in server;
      struct hostent *hp, *gethostbyname();
      char buf[1024];

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sin_family = AF_INET;
      hp = gethostbyname(argv[1]);
      if (hp == 0) {
            fprintf(stderr, "%s: unknown host0, argv[1]);
            exit(2);
      }
      bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
      server.sin_port = htons(atoi(argv[2]));

      if (connect(sock, &server, sizeof(server)) < 0) {
            perror("connecting stream socket");
            exit(1);
      }
      if (write(sock, DATA, sizeof(DATA)) < 0)
            perror("writing on stream socket");
      close(sock);
}
```

Figure 7a Initiating an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each time
 * through the loop it accepts a connection and prints out messages from it.
 * When the connection breaks, or a termination message comes through, the
 * program accepts a new connection.
 */

main()
{
      int sock, length;
      struct sockaddr_in server;
      int msgsock;
      char buf[1024];
      int rval;
      int i;

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using wildcards */
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = INADDR_ANY;
      server.sin_port = 0;
      if (bind(sock, &server, sizeof(server))) {
            perror("binding stream socket");
            exit(1);
      }
      /* Find out assigned port number and print it out */
      length = sizeof(server);
      if (getsockname(sock, &server, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(server.sin_port));

      /* Start accepting connections */
      listen(sock, 5);
      do {
            msgsock = accept(sock, 0, 0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
```

```
                        if ((rval = read(msgsock, buf, 1024)) < 0)
                              perror("reading stream message");
                        i = 0;
                        if (rval == 0)
                              printf("Ending connection\n");
                        else
                              printf("-->%s\n", buf);
                  } while (rval != 0);
                  close(msgsock);
            } while (TRUE);
            /*
             * Since this program has an infinite loop, the socket "sock" is
             * never explicitly closed.  However, all sockets will be closed
             * automatically when a process is killed or terminates normally.
             */
}
```

Figure 7b Accepting an Internet domain stream connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program uses select() to check that someone is trying to connect
 * before calling accept().
 */

main()
{
      int sock, length;
      struct sockaddr_in server;
      int msgsock;
      char buf[1024];
      int rval;
      fd_set ready;
      struct timeval to;

      /* Create socket */
      sock = socket(AF_INET, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using wildcards */
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = INADDR_ANY;
      server.sin_port = 0;
      if (bind(sock, &server, sizeof(server))) {
            perror("binding stream socket");
            exit(1);
      }
      /* Find out assigned port number and print it out */
      length = sizeof(server);
      if (getsockname(sock, &server, &length)) {
            perror("getting socket name");
            exit(1);
      }
      printf("Socket has port #%d\n", ntohs(server.sin_port));

      /* Start accepting connections */
      listen(sock, 5);
      do {
            FD_ZERO(&ready);
            FD_SET(sock, &ready);
            to.tv_sec = 5;
            if (select(sock + 1, &ready, 0, 0, &to) < 0) {
                  perror("select");
                  continue;
            }
            if (FD_ISSET(sock, &ready)) {
                  msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
                  if (msgsock == -1)
                        perror("accept");
                  else do {
                        bzero(buf, sizeof(buf));
                        if ((rval = read(msgsock, buf, 1024)) < 0)
                              perror("reading stream message");
                        else if (rval == 0)
                              printf("Ending connection\n");
                        else
                              printf("-->%s\n", buf);
                  } while (rval > 0);
                  close(msgsock);
            } else
                  printf("Do something else\n");
      } while (TRUE);
}
```

Figure 7c Using select() to check for pending connections

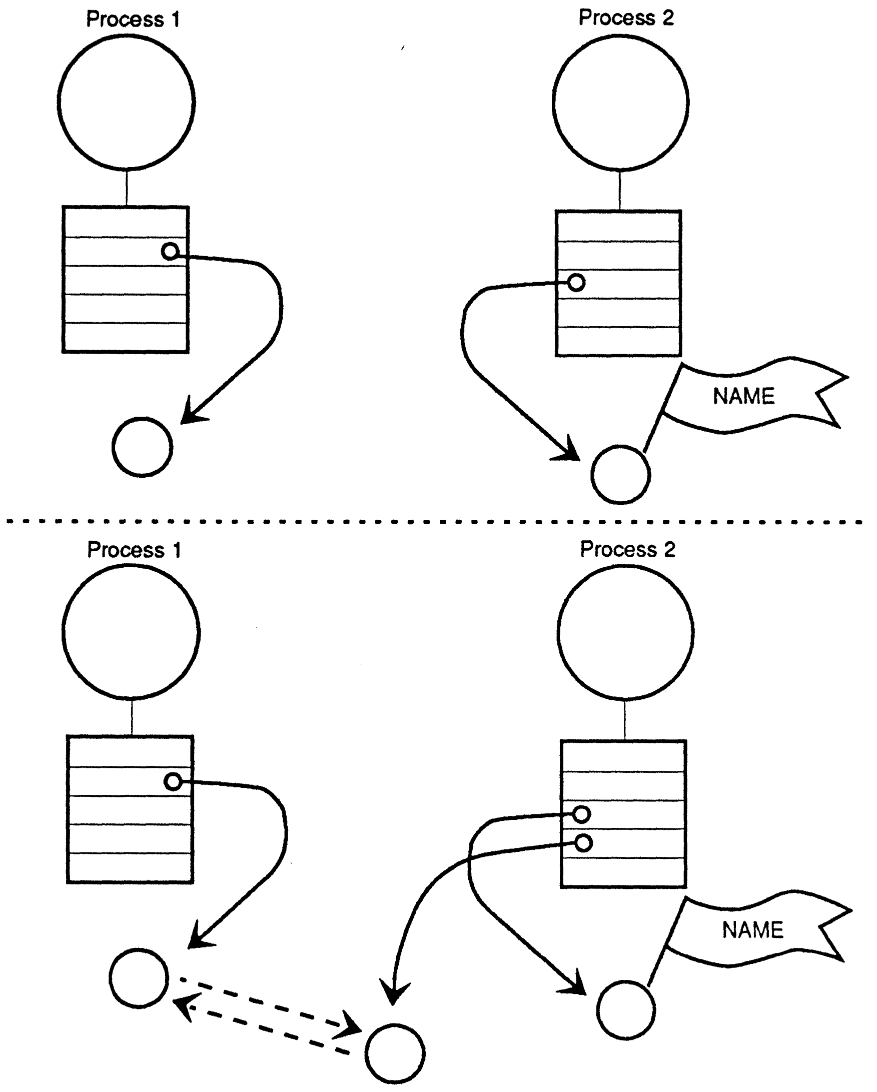

Figure 8 Establishing a stream connection

call has been completed, the program enters an infinite loop. On each pass through the loop, a new connection is accepted and removed from the queue, and, hence, a new socket for the connection is created. The bottom half of Figure 8 shows the result of Process 1 connecting with the named socket of Process 2, and Process 2 accepting the connection. After the connection is created, the service, in this case printing out the messages, is performed and the connection socket closed. The *accept()* call will take a pending connection request from the queue if one is available, or block waiting for a request. Messages are read from the connection socket. Reads from an active connection will normally block until data is available. The number of bytes read is returned. When a connection is destroyed, the read call returns immediately. The number of bytes returned will be zero.

The program in Figure 7c is a slight variation on the server in Figure 7b. It avoids blocking when there are no pending connection requests by calling *select()* to check for pending requests before calling *accept()*. This strategy is useful when connections may be received on more than one socket, or when data may arrive on other connected sockets before another connection request.

The programs in Figures 9a and 9b show a program using stream communication in the UNIX domain. Streams in the UNIX domain can be used for this sort of program in exactly the same way as Internet domain streams, except for the form of the names and the restriction of the connections to a single file system. There are some differences, however, in the functionality of streams in the two domains, notably in the handling of *out-of-band* data (discussed briefly below). These differences are beyond the scope of this paper.

9. Reads, Writes, Recvs, etc.

UNIX 4.3BSD has several system calls for reading and writing information. The simplest calls are *read()* and *write()*. *Write()* takes as arguments the index of a descriptor, a pointer to a buffer containing the data and the size of the data. The descriptor may indicate either a file or a connected socket. ''Connected'' can mean either a connected stream socket (as described in Section 8) or a datagram socket for which a *connect()* call has provided a default destination (see the *connect()* manual page). *Read()* also takes a descriptor that indicates either a file or a socket. *Write()* requires a connected socket since no destination is specified in the parameters of the system call. *Read()* can be used for either a connected or an unconnected socket. These calls are, therefore, quite flexible and may be used to write applications that require no assumptions about the source of their input or the destination of their output. There are variations on *read()* and *write()* that allow the source and destination of the input and output to use several separate buffers, while retaining the flexibility to handle both files and sockets. These are *readv()* and *writev()*, for read and write *vector*.

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. For example, a user interface process may be interpreting commands and sending them on to another process through a stream connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to send it as *out-of-band* (OOB) data. The notification of pending OOB data results in the generation of a SIGURG signal, if this signal has been enabled (see the manual page for *signal* or *sigvec*). See [Leffler 1986] for a more complete description of the OOB mechanism. There are a pair of calls similar to *read* and *write* that allow options, including sending and receiving OOB information; these are *send()* and *recv()*. These calls are used only with sockets; specifying a descriptor for a file will result in the return of an error status. These calls also allow *peeking* at data in a stream. That is, they allow a process to read data without removing the data from the stream. One use of this facility is to read ahead in a stream to determine the size of the next item to be read. When not using these options, these calls have the same functions as *read()* and *write()*.

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

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

The various options for reading and writing are shown in Figure 10, together with their parameters. The parameters for each system call reflect the differences in function of the different calls. In the examples given in this paper, the calls *read()* and *write()* have been used whenever possible.

10. Choices

This paper has presented examples of some of the forms of communication supported by Berkeley UNIX 4.3BSD. These have been presented in an order chosen for ease of presentation. It is useful to review these options emphasizing the factors that make each attractive.

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

#define DATA "Half a league, half a league . . ."

/*
 * This program connects to the socket named in the command line and sends a
 * one line message to that socket. The form of the command line is
 * ustreamwrite pathname
 */
main(argc, argv)
      int argc;
      char *argv[];
{
      int sock;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Connect socket using name specified by command line. */
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, argv[1]);

      if (connect(sock, &server, sizeof(struct sockaddr_un)) < 0) {
            close(sock);
            perror("connecting stream socket");
            exit(1);
      }
      if (write(sock, DATA, sizeof(DATA)) < 0)
            perror("writing on stream socket");
}
```

Figure 9a Initiating a UNIX domain stream connection

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

#define NAME "socket"

/*
 * This program creates a socket in the UNIX domain and binds a name to it.
 * After printing the socket's name it begins a loop. Each time through the
 * loop it accepts a connection and prints out messages from it.  When the
 * connection breaks, or a termination message comes through, the program
 * accepts a new connection.
 */
main()
{
      int sock, msgsock, rval;
      struct sockaddr_un server;
      char buf[1024];

      /* Create socket */
      sock = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sock < 0) {
            perror("opening stream socket");
            exit(1);
      }
      /* Name socket using file system name */
      server.sun_family = AF_UNIX;
      strcpy(server.sun_path, NAME);
      if (bind(sock, &server, sizeof(struct sockaddr_un))) {
            perror("binding stream socket");
            exit(1);
      }
      printf("Socket has name %s\n", server.sun_path);
      /* Start accepting connections */
      listen(sock, 5);
      for (;;) {
            msgsock = accept(sock, 0, 0);
            if (msgsock == -1)
                  perror("accept");
            else do {
                  bzero(buf, sizeof(buf));
                  if ((rval = read(msgsock, buf, 1024)) < 0)
                        perror("reading stream message");
                  else if (rval == 0)
                        printf("Ending connection\n");
                  else
                        printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
      }
      /*
       * The following statements are not executed, because they follow an
       * infinite loop.  However, most ordinary programs will not run
       * forever.  In the UNIX domain it is necessary to tell·the file
       * system that one is through using NAME.  In most programs one uses
       * the call unlink() as below. Since the user will have to kill this
       * program, it will be necessary to remove the name by a command from
       * the shell.
       */
      close(sock);
      unlink(NAME);
}
```

Figure 9b Accepting a UNIX domain stream connection

```
/*
 * The variable descriptor may be the descriptor of either a file
 * or of a socket.
 */
cc = read(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

/*
 * An iovec can include several source buffers.
 */
cc = readv(descriptor, iov, iovcnt)
int cc, descriptor; struct iovec *iov; int iovcnt;

cc = write(descriptor, buf, nbytes)
int cc, descriptor; char *buf; int nbytes;

cc = writev(descriptor, iovec, ioveclen)
int cc, descriptor; struct iovec *iovec; int ioveclen;

/*
 * The variable ''sock'' must be the descriptor of a socket.
 * Flags may include MSG_OOB and MSG_PEEK.
 */
cc = send(sock, msg, len, flags)
int cc, sock; char *msg; int len, flags;

cc = sendto(sock, msg, len, flags, to, tolen)
int cc, sock; char *msg; int len, flags;
struct sockaddr *to; int tolen;

cc = sendmsg(sock, msg, flags)
int cc, sock; struct msghdr msg[]; int flags;

cc = recv(sock, buf, len, flags)
int cc, sock; char *buf; int len, flags;

cc = recvfrom(sock, buf, len, flags, from, fromlen)
int cc, sock; char *buf; int len, flags;
struct sockaddr *from; int *fromlen;

cc = recvmsg(sock, msg, flags)
int cc, socket; struct msghdr msg[]; int flags;
```

Figure 10 Varieties of read and write commands

Pipes have the advantage of portability, in that they are supported in all UNIX systems. They also are relatively simple to use. Socketpairs share this simplicity and have the additional advantage of allowing bidirectional communication. The major shortcoming of these mechanisms is that they require communicating processes to be descendants of a common process. They do not allow intermachine communication.

The two communication domains, UNIX and Internet, allow processes with no common ancestor to communicate. Of the two, only the Internet domain allows communication between machines. This makes the Internet domain a necessary choice for processes running on separate machines.

The choice between datagrams and stream communication is best made by carefully considering the semantic and performance requirements of the application. Streams can be both advantageous and disadvantageous. One disadvantage is that a process is only allowed a limited number of open streams, as there are usually only 64 entries available in the open descriptor table. This can cause problems if a single server must talk with a large number of clients. Another is that for delivering a short message the stream setup and teardown time can be unnecessarily long. Weighed against this are the reliability built into the streams. This will often be the deciding factor in favor of streams.

11. What to do Next

Many of the examples presented here can serve as models for multiprocess programs and for programs distributed across several machines. In developing a new multiprocess program, it is often easiest to first write the code to create the processes and communication paths. After this code is debugged, the code specific to the application can be added.

An introduction to the UNIX system and programming using UNIX system calls can be found in [Kernighan and Pike 1984]. Further documentation of the Berkeley UNIX 4.3BSD IPC mechanisms can be found in [Leffler et al. 1986]. More detailed information about particular calls and protocols is provided in sections 2, 3 and 4 of the UNIX Programmer's Manual [CSRG 1986]. In particular the following manual pages are relevant:

creating and naming sockets	socket(2), bind(2)
establishing connections	listen(2), accept(2), connect(2)
transferring data	read(2), write(2), send(2), recv(2)
addresses	inet(4F)
protocols	tcp(4P), udp(4P).

Acknowledgements

I would like to thank Sam Leffler and Mike Karels for their help in understanding the IPC mechanisms and all the people whose comments have helped in writing and improving this report.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

References

B.W. Kernighan & R. Pike, 1984,
The UNIX Programming Environment.
Englewood Cliffs, N.J.: Prentice-Hall.

B.W. Kernighan & D.M. Ritchie, 1978,
The C Programming Language,
Englewood Cliffs, N.J.: Prentice-Hall.

S.J. Leffler, R.S. Fabry, W.N. Joy, P. Lapsley, S. Miller & C. Torek, 1986,
An Advanced 4.3BSD Interprocess Communication Tutorial.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

Computer Systems Research Group, 1986,
UNIX Programmer's Manual, 4.3 Berkeley Software Distribution.
Computer Systems Research Group,
Department of Electrical Engineering and Computer Science,
University of California, Berkeley.

Appendix G – An Advanced Interprocess Communication Tutorial

The following appendix contains a document produced by Samuel J. Leffler, Robert S. Fabry, William N. Joy, and Phil Lapsley of the Computer Systems Research Group, Department of Electrical Engineering and Computer Science at the University of California, Berkeley, and Steve Miller and Chris Torek of the Heterogeneous Systems Laboratory, Department of Computer Science at the University of Maryland, College Park, Maryland, titled "*An Advanced 4.3BSD Interprocess Communication Tutorial*". The document provides an introduction to the interprocess communication facilities included in the 4.3BSD release of the UNIX® system. It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system.

An Advanced 4.3BSD Interprocess Communication Tutorial

Samuel J. Leffler

Robert S. Fabry

William N. Joy

Phil Lapsley

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

Steve Miller

Chris Torek

Heterogeneous Systems Laboratory
Department of Computer Science
University of Maryland, College Park
College Park, Maryland 20742

ABSTRACT

This document provides an introduction to the interprocess communication facilities included in the 4.3BSD release of the UNIX* system.

It discusses the overall model for interprocess communication and introduces the interprocess communication primitives which have been added to the system. The majority of the document considers the use of these primitives in developing applications. The reader is expected to be familiar with the C programming language as all examples are written in C.

* UNIX is a Trademark of Bell Laboratories.

1. INTRODUCTION

One of the most important additions to UNIX in 4.2BSD was interprocess communication. These facilities were the result of more than two years of discussion and research. The facilities provided in 4.2BSD incorporated many of the ideas from current research, while trying to maintain the UNIX philosophy of simplicity and conciseness. The current release of Berkeley UNIX, 4.3BSD, completes some of the IPC facilities and provides an upward-compatible interface. It is hoped that the interprocess communication facilities included in 4.3BSD will establish a standard for UNIX. From the response to the design, it appears many organizations carrying out work with UNIX are adopting it.

UNIX has previously been very weak in the area of interprocess communication. Prior to the 4BSD facilities, the only standard mechanism which allowed two processes to communicate were pipes (the mpx files which were part of Version 7 were experimental). Unfortunately, pipes are very restrictive in that the two communicating processes must be related through a common ancestor. Further, the semantics of pipes makes them almost impossible to maintain in a distributed environment.

Earlier attempts at extending the IPC facilities of UNIX have met with mixed reaction. The majority of the problems have been related to the fact that these facilities have been tied to the UNIX file system, either through naming or implementation. Consequently, the IPC facilities provided in 4.3BSD have been designed as a totally independent subsystem. The 4.3BSD IPC allows processes to rendezvous in many ways. Processes may rendezvous through a UNIX file system-like name space (a space where all names are path names) as well as through a network name space. In fact, new name spaces may be added at a future time with only minor changes visible to users. Further, the communication facilities have been extended to include more than the simple byte stream provided by a pipe. These extensions have resulted in a completely new part of the system which users will need time to familiarize themselves with. It is likely that as more use is made of these facilities they will be refined; only time will tell.

This document provides a high-level description of the IPC facilities in 4.3BSD and their use. It is designed to complement the manual pages for the IPC primitives by examples of their use. The remainder of this document is organized in four sections. Section 2 introduces the IPC-related system calls and the basic model of communication. Section 3 describes some of the supporting library routines users may find useful in constructing distributed applications. Section 4 is concerned with the client/server model used in developing applications and includes examples of the two major types of servers. Section 5 delves into advanced topics which sophisticated users are likely to encounter when using the IPC facilities.

2. BASICS

The basic building block for communication is the *socket*. A socket is an endpoint of communication to which a name may be *bound*. Each socket in use has a *type* and one or more associated processes. Sockets exist within *communication domains*. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. One such property is the scheme used to name sockets. For example, in the UNIX communication domain sockets are named with UNIX path names; e.g. a socket may be named "/dev/foo". Sockets normally exchange data only with sockets in the same domain (it may be possible to cross domain boundaries, but only if some translation process is performed). The 4.3BSD IPC facilities support three separate communication domains: the UNIX domain, for on-system communication; the Internet domain, which is used by processes which communicate using the the DARPA standard communication protocols; and the NS domain, which is used by processes which communicate using the Xerox standard communication protocols*. The underlying communication facilities provided by these domains have a significant influence on the internal system implementation as well as the interface to socket facilities available to a user. An example of the latter is that a socket "operating" in the UNIX domain sees a subset of the error conditions which are possible when operating in the Internet (or NS) domain.

2.1. Socket types

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

Four types of sockets currently are available to a user. A *stream* socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries. Aside from the bidirectionality of data flow, a pair of connected stream sockets provides an interface nearly identical to that of pipes†.

A *datagram* socket supports bidirectional flow of data which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved. Datagram sockets closely model the facilities found in many contemporary packet switched networks such as the Ethernet.

A *raw* socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. The use of raw sockets is considered in section 5.

A *sequenced packet* socket is similar to a stream socket, with the exception that record boundaries are preserved. This interface is provided only as part of the NS socket abstraction, and is very important in most serious NS applications. Sequenced-packet sockets allow the user to manipulate the SPP or IDP headers on a packet or a group of packets either by writing a prototype header along with whatever data is to be sent, or by specifying a default header to be used with all outgoing data, and allows the user to receive the headers on incoming packets. The use of these options is considered in section 5.

Another potential socket type which has interesting properties is the *reliably delivered message* socket. The reliably delivered message socket has similar properties to a datagram socket, but with

* See *Internet Transport Protocols*, Xerox System Integration Standard (XSIS)028112 for more information. This document is almost a necessity for one trying to write NS applications.

† In the UNIX domain, in fact, the semantics are identical and, as one might expect, pipes have been implemented internally as simply a pair of connected stream sockets.

reliable delivery. There is currently no support for this type of socket, but a reliably delivered message protocol similar to Xerox's Packet Exchange Protocol (PEX) may be simulated at the user level. More information on this topic can be found in section 5.

2.2. Socket creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified *domain* and of the specified *type*. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type. The user is returned a descriptor (a small integer number) which may be used in later system calls which operate on sockets. The domain is specified as one of the manifest constants defined in the file <*sys/socket.h*>. For the UNIX domain the constant is AF_UNIX*; for the Internet domain AF_INET; and for the NS domain, AF_NS. The socket types are also defined in this file and one of SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, or SOCK_SEQPACKET must be specified. To create a stream socket in the Internet domain the following call might be used:

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

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket for on-machine use the call might be:

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

The default protocol (used when the *protocol* argument to the *socket* call is 0) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this will be covered in section 5.

There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

2.3. Binding local names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it. Communicating processes are bound by an *association*. In the Internet and NS domains, an association is composed of local and foreign addresses, and local and foreign ports, while in the UNIX domain, an association is composed of local and foreign path names (the phrase "foreign pathname" means a pathname created by a foreign process, not a pathname on a foreign system). In most domains, associations must be unique. In the Internet domain there may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples. UNIX domain sockets need not always be bound to a name, but when bound there may never be duplicate <protocol, local pathname, foreign pathname> tuples. The pathnames may not refer to files already existing on the system in 4.3; the situation may change in future releases.

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

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

The *bind* system call is used as follows:

* The manifest constants are named AF_whatever as they indicate the "address format" to use in interpreting names.

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

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). Its interpretation may vary from communication domain to communication domain (this is one of the properties which comprise the "domain"). As mentioned, in the Internet domain names contain an Internet address and port number. NS domain names contain an NS address and port number. In the UNIX domain, names contain a path name and a family, which is always AF_UNIX. If one wanted to bind the name "/tmp/foo" to a UNIX domain socket, the following code would be used*:

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

Note that in determining the size of a UNIX domain address null bytes are not counted, which is why *strlen* is used. In the current implementation of UNIX domain IPC under 4.3BSD, the file name referred to in *addr.sun_path* is created as a socket in the system file space. The caller must, therefore, have write permission in the directory where *addr.sun_path* is to reside, and this file should be deleted by the caller when it is no longer needed. Future versions of 4BSD may not create this file.

In binding an Internet address things become more complicated. The actual call is similar,

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

but the selection of what to place in the address *sin* requires some discussion. We will come back to the problem of formulating Internet addresses in section 3 when the library routines used in name resolution are discussed.

Binding an NS address to a socket is even more difficult, especially since the Internet library routines do not work with NS hostnames. The actual call is again similar:

```
#include <sys/types.h>
#include <netns/ns.h>
 ...
struct sockaddr_ns sns;
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

Again, discussion of what to place in a "struct sockaddr_ns" will be deferred to section 3.

2.4. Connection establishment

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

* Note that, although the tendency here is to call the "addr" structure "sun", doing so would cause problems if the code were ever ported to a Sun workstation.

the UNIX domain, this might appear as,

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

while in the Internet domain,

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

and in the NS domain,

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

where *server* in the example above would contain either the UNIX pathname, Internet address and port number, or NS address and port number of the server to which the client process wishes to speak. If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary; c.f. section 5.4. This is the usual way that local addresses are bound to a socket.

An error is returned if the connection was unsuccessful (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin. Some of the more common errors returned when a connection attempt fails are:

ETIMEDOUT
: After failing to establish a connection for a period of time, the system decided there was no point in retrying the connection attempt any more. This usually occurs because the destination host is down, or because problems in the network resulted in transmissions being lost.

ECONNREFUSED
: The host refused service for some reason. This is usually due to a server process not being present at the requested name.

ENETDOWN or EHOSTDOWN
: These operational errors are returned based on status information delivered to the client host by the underlying communication services.

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

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

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

The second parameter to the *listen* call specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process; this number may be limited by the system. Should a connection be requested while the queue is full, the connection will not be refused, but rather the individual messages which comprise the request will be ignored. This gives a harried server time to make room in its pending connection queue while the client retries the connection request. Had the connection been returned with the ECONNREFUSED error, the client would be unable to tell if the server was up or not. As it is now it is still possible to get the ETIMEDOUT error back, though this is unlikely. The backlog figure supplied with the listen call is currently limited by the system to a maximum of 5 pending connections on any one queue. This avoids the problem of processes hogging system resources by setting an infinite backlog, then ignoring all connection requests.

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

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

(For the UNIX domain, *from* would be declared as a *struct sockaddr_un*, and for the NS domain, *from* would be declared as a *struct sockaddr_ns*, but nothing different would need to be done as far as *fromlen* is concerned. In the examples which follow, only Internet routines will be discussed.) A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name. The value-result parameter *fromlen* is initialized by the server to indicate how much space is associated with *from*, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

Accept normally blocks. That is, *accept* will not return until a connection is available or the system call is interrupted by a signal to the process. Further, there is no way for a process to indicate it will accept connections from only a specific individual, or individuals. It is up to the user process to consider who the connection is from and close down the connection if it does not wish to speak to the process. If the server process wants to accept connections on more than one socket, or wants to avoid blocking on the accept call, there are alternatives; they will be considered in section 5.

2.5. Data transfer

With a connection established, data may begin to flow. To send and receive data there are a number of possible calls. With the peer entity at each end of a connection anchored, a user can send or receive a message without specifying the peer. As one might expect, in this case, then the normal *read* and *write* system calls are usable,

```
write(s, buf, sizeof (buf));
read(s, buf, sizeof (buf));
```

In addition to *read* and *write*, the new calls *send* and *recv* may be used:

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

While *send* and *recv* are virtually identical to *read* and *write*, the extra *flags* argument is important. The flags, defined in <*sys/socket.h*>, may be specified as a non-zero value if one or more of the following is required:

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

Out of band data is a notion specific to stream sockets, and one which we will not immediately consider. The option to have data sent without routing applied to the outgoing packets is currently used only by the routing table management process, and is unlikely to be of interest to the casual user. The ability to preview data is, however, of interest. When MSG_PEEK is specified with a *recv* call, any data present is returned to the user, but treated as still ``unread''. That is, the next *read* or *recv* call applied to the socket will return the data previously previewed.

2.6. Discarding sockets

Once a socket is no longer of interest, it may be discarded by applying a *close* to the descriptor,

```
close(s);
```

If data is associated with a socket which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded. Should a user have no use for any pending data, it may perform a *shutdown* on the socket prior to closing it. This call is of the form:

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

where *how* is 0 if the user is no longer interested in reading data, 1 if no more data will be sent, or 2 if no data is to be sent or received.

2.7. Connectionless sockets

To this point we have been concerned mostly with sockets which follow a connection oriented model. However, there is also support for connectionless interactions typical of the datagram facilities found in contemporary packet switched networks. A datagram socket provides a symmetric interface to data exchange. While processes are still likely to be client and server, there is no requirement for connection establishment. Instead, each message includes the destination address.

Datagram sockets are created as before. If a particular local address is needed, the *bind* operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent. To send data, the *sendto* primitive is used,

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

The *s*, *buf*, *buflen*, and *flags* parameters are used as before. The *to* and *tolen* values are used to indicate the address of the intended recipient of the message. When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When information is present locally to recognize a message that can not be delivered (for instance when a network is unreachable), the call will return -1 and the global value *errno* will contain an error number.

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

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

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

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

While a datagram socket socket is connected, errors from recent *send* calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket, or a special socket option used with *getsockopt*, SO_ERROR, may be used to interrogate the error status. A *select* for reading or writing will return true when an error indication has been received. The next operation will return the error, and the error status is cleared. Other of the less important details of datagram sockets are described in section 5.

2.8. Input/Output multiplexing

One last facility often used in developing applications is the ability to multiplex i/o requests among multiple sockets and/or files. This is done using the *select* call:

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

fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

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

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

The macros FD_SET(*fd, &mask*) and FD_CLR(*fd, &mask*) have been provided for adding and removing file descriptor *fd* in the set *mask*. The set should be zeroed before use, and the macro FD_ZERO(*&mask*) has been provided to clear the set *mask*. The parameter *nfds* in the *select* call specifies the range of file descriptors (i.e. one plus the value of the largest descriptor) to be examined in a set.

A timeout value may be specified if the selection is not to last more than a predetermined period of time. If the fields in *timeout* are set to 0, the selection takes the form of a *poll*, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely*. *Select* normally returns the number of file descriptors selected; if the *select* call returns due to the timeout expiring, then the value 0 is returned. If the *select* terminates because of an error or interruption, a -1 is returned with the error number in *errno*, and with the file descriptor masks unchanged.

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

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

As an example, to read data from two sockets, *s1* and *s2* as it is available from each and with a one-second timeout, the following code might be used:

* To be more specific, a return takes place only when a descriptor is selectable, or when a signal is received by the caller, interrupting the system call.

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

        FD_ZERO(&read_template);

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

        nb = select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
        if (nb <= 0) {
                An error occurred during the select, or
                the select timed out.
        }

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

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

In 4.2, the arguments to *select* were pointers to integers instead of pointers to *fd_sets*. This type of call will still work as long as the number of file descriptors being examined is less than the number of bits in an integer; however, the methods illustrated above should be used in all current programs.

Select provides a synchronous multiplexing scheme. Asynchronous notification of output completion, input availability, and exceptional conditions is possible through use of the SIGIO and SIGURG signals described in section 5.

3. NETWORK LIBRARY ROUTINES

The discussion in section 2 indicated the possible need to locate and construct network addresses when using the interprocess communication facilities in a distributed environment. To aid in this task a number of routines have been added to the standard C run-time library. In this section we will consider the new routines provided to manipulate network addresses. While the 4.3BSD networking facilities support both the DARPA standard Internet protocols and the Xerox NS protocols, most of the routines presented in this section do not apply to the NS domain. Unless otherwise stated, it should be assumed that the routines presented in this section do not apply to the NS domain.

Locating a service on a remote host requires many levels of mapping before client and server may communicate. A service is assigned a name which is intended for human consumption; e.g. ''the *login server* on host monet''. This name, and the name of the peer host, must then be translated into network *addresses* which are not necessarily suitable for human consumption. Finally, the address must then used in locating a physical *location* and *route* to the service. The specifics of these three mappings are likely to vary between network architectures. For instance, it is desirable for a network to not require hosts to be named in such a way that their physical location is known by the client host. Instead, underlying services in the network may discover the actual location of the host at the time a client host wishes to communicate. This ability to have hosts named in a location independent manner may induce overhead in connection establishment, as a discovery process must take place, but allows a host to be physically mobile without requiring it to notify its clientele of its current location.

Standard routines are provided for: mapping host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers and the appropriate protocol to use in communicating with the server process. The file <*netdb.h*> must be included when using any of these routines.

3.1. Host names

An Internet host name to address mapping is represented by the *hostent* structure:

```
struct    hostent {
          char     *h_name;            /* official name of host */
          char     **h_aliases;        /* alias list */
          int      h_addrtype;         /* host address type (e.g., AF_INET) */
          int      h_length;           /* length of address */
          char     **h_addr_list;      /* list of addresses, null terminated */
};

#define   h_addr   h_addr_list[0]      /* first address, network byte order */
```

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

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

The database for these calls is provided either by the file */etc/hosts* (*hosts*(5)), or by use of a nameserver, *named*(8). Because of the differences in these databases and their access protocols, the information returned may differ. When using the host table version of *gethostbyname*, only one address will be returned, but all listed aliases will be included. The nameserver version may return alternate addresses, but will not provide any aliases other than one given as argument.

Unlike Internet names, NS names are always mapped into host addresses by the use of a standard NS *Clearinghouse service*, a distributed name and authentication server. The algorithms for mapping NS names to addresses via a Clearinghouse are rather complicated, and the routines are not part of the standard libraries. The user-contributed Courier (Xerox remote procedure call protocol) compiler contains routines to accomplish this mapping; see the documentation and examples provided therein for more information. It is expected that almost all software that has to communicate using NS will need to use the facilities of the Courier compiler.

An NS host address is represented by the following:

```
union ns_host {
        u_char      c_host[6];
        u_short     s_host[3];
};

union ns_net {
        u_char      c_net[4];
        u_short     s_net[2];
};

struct ns_addr {
        union ns_net        x_net;
        union ns_host       x_host;
        u_short     x_port;
};
```

The following code fragment inserts a known NS address into a *ns_addr*:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
u_long netnum;
struct sockaddr_ns dst;
 ...
bzero((char *)&dst, sizeof(dst));

/*
 * There is no convenient way to assign a long
 * integer to a ''union ns_net'' at present; in
 * the future, something will hopefully be provided,
 * but this is the portable way to go for now.
 * The network number below is the one for the NS net
 * that the desired host (gyre) is on.
 */
netnum = htonl(2266);
dst.sns_addr.x_net = *(union ns_net *) &netnum;
dst.sns_family = AF_NS;

/*
 * host 2.7.1.0.2a.18 == "gyre:Computer Science:UofMaryland"
 */
dst.sns_addr.x_host.c_host[0] = 0x02;
dst.sns_addr.x_host.c_host[1] = 0x07;
dst.sns_addr.x_host.c_host[2] = 0x01;
dst.sns_addr.x_host.c_host[3] = 0x00;
dst.sns_addr.x_host.c_host[4] = 0x2a;
dst.sns_addr.x_host.c_host[5] = 0x18;
dst.sns_addr.x_port = htons(75);
```

3.2. Network names

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

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

The routines *getnetbyname*(3N), *getnetbynumber*(3N), and *getnetent*(3N) are the network counterparts to the host routines described above. The routines extract their information from */etc/networks.*

NS network numbers are determined either by asking your local Xerox Network Administrator (and hardcoding the information into your code), or by querying the Clearinghouse for addresses. The internetwork router is the only process that needs to manipulate network numbers on a regular basis; if a process wishes to communicate with a machine, it should ask the Clearinghouse for that machine's address (which will include the net number).

3.3. Protocol names

For protocols, which are defined in */etc/protocols*, the *protoent* structure defines the protocol-name mapping used with the routines *getprotobyname*(3N), *getprotobynumber*(3N), and *getprotoent*(3N):

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

In the NS domain, protocols are indicated by the "client type" field of a IDP header. No protocol database exists; see section 5 for more information.

3.4. Service names

Information regarding services is a bit more complicated. A service is expected to reside at a specific ''port'' and employ a particular communication protocol. This view is consistent with the Internet domain, but inconsistent with other network architectures. Further, a service may reside on multiple ports. If this occurs, the higher level library routines will have to be bypassed or extended. Services available are contained in the file */etc/services*. A service mapping is described by the *servent* structure,

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

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

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

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

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

returns only that telnet server which uses the TCP protocol. The routines *getservbyport*(3N) and *getservent*(3N) are also provided. The *getservbyport* routine has an interface similar to that provided by *getservbyname*; an optional protocol name may be specified to qualify lookups.

In the NS domain, services are handled by a central dispatcher provided as part of the Courier remote procedure call facilities. Again, the reader is referred to the Courier compiler documentation and to the Xerox standard* for further details.

3.5. Miscellaneous

With the support routines described above, an Internet application program should rarely have to deal directly with addresses. This allows services to be developed as much as possible in a network independent fashion. It is clear, however, that purging all network dependencies is very difficult. So long as the user is required to supply network addresses when naming services and sockets there will always some network dependency in a program. For example, the normal code included in client programs, such as the remote login program, is of the form shown in Figure 1. (This example will be considered in more detail in section 4.)

If we wanted to make the remote login program independent of the Internet protocols and addressing scheme we would be forced to add a layer of routines which masked the network dependent aspects from the mainstream login code. For the current facilities available in the system this does not appear to be

* *Courier: The Remote Procedure Call Protocol*, XSIS 038112.

worthwhile.

Aside from the address-related data base routines, there are several other routines available in the run-time library which are of interest to users. These are intended mostly to simplify manipulation of names and addresses. Table 1 summarizes the routines for manipulating variable length byte strings and handling byte swapping of network addresses and values.

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

Table 1. C run-time routines.

The byte swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, such as the VAX, host byte ordering is different than network byte ordering. Consequently, programs are sometimes required to byte swap quantities. The library routines which return network addresses provide them in network order so that they may simply be copied into the structures provided to the system. This implies users should encounter the byte swapping problem only when *interpreting* network addresses. For example, if an Internet port is to be printed out the following code would be required:

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

On machines where unneeded these routines are defined as null macros.

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

        if (connect(s, (char *)&server, sizeof (server)) < 0) {
                perror("rlogin: connect");
                exit(5);
        }
        ...
}
```

Figure 1. Remote login client code.

4. CLIENT/SERVER MODEL

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server process. This implies an asymmetry in establishing communication between the client and server which has been examined in section 2. In this section we will look more closely at the interactions between client and server, and consider some of the problems in developing client and server applications.

The client and server require a well known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol which must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is immutably recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a ''client process'' and a ''server process''. We will first consider the properties of server processes, then client processes.

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

Alternative schemes which use a service server may be used to eliminate a flock of server processes clogging the system while remaining dormant most of the time. For Internet servers in 4.3BSD, this scheme has been implemented via *inetd*, the so called ''internet super-server.'' *Inetd* listens at a variety of ports, determined at start-up by reading a configuration file. When a connection is requested to a port on which *inetd* is listening, *inetd* executes the appropriate server program to handle the client. With this method, clients are unaware that an intermediary such as *inetd* has played any part in the connection. *Inetd* will be described in more detail in section 5.

A similar alternative scheme is used by most Xerox services. In general, the Courier dispatch process (if used) accepts connections from processes requesting services of some sort or another. The client processes request a particular <program number, version number, procedure number> triple. If the dispatcher knows of such a program, it is started to handle the request; if not, an error is reported to the client. In this way, only one port is required to service a large variety of different requests. Again, the Courier facilities are not available without the use and installation of the Courier compiler. The information presented in this section applies only to NS clients and services that do not use Courier.

4.1. Servers

In 4.3BSD most servers are accessed at well known Internet addresses or UNIX domain names. For example, the remote login server's main loop is of the form shown in Figure 2.

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

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

The result of the *getservbyname* call is used in later portions of the code to define the Internet port at which it listens for service requests (indicated by a connection).

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

	sp = getservbyname("login", "tcp");
	if (sp == NULL) {
		fprintf(stderr, "rlogind: tcp/login: unknown service\n");
		exit(1);
	}
	...
#ifndef DEBUG
	/* Disassociate server from controlling terminal */
	...
#endif

	sin.sin_port = sp->s_port; /* Restricted port -- see section 5 */
	...
	f = socket(AF_INET, SOCK_STREAM, 0);
	...
	if (bind(f, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
		...
	}
	...
	listen(f, 5);
	for (;;) {
		int g, len = sizeof (from);

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

Figure 2. Remote login server.

Step two is to disassociate the server from the controlling terminal of its invoker:

```
for (i = 0; i < 3; ++i)
      close(i);

open("/", O_RDONLY);
dup2(0, 1);
dup2(0, 2);

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

This step is important as the server will likely not want to receive signals delivered to the process group of the controlling terminal. Note, however, that once a server has disassociated itself it can no longer send reports of errors to a terminal, and must log errors via *syslog*.

Once a server has established a pristine environment, it creates a socket and begins accepting service requests. The *bind* call is required to insure the server listens at its expected location. It should be noted that the remote login server listens at a restricted port number, and must therefore be run with a user-id of root. This concept of a ‘‘restricted port number’’ is 4BSD specific, and is covered in section 5.

The main body of the loop is fairly simple:

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

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

An *accept* call blocks the server until a client requests service. This call could return a failure status if the call is interrupted by a signal such as SIGCHLD (to be discussed in section 5). Therefore, the return value from *accept* is checked to insure a connection has actually been established, and an error report is logged via *syslog* if an error has occurred.

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

4.2. Clients

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

Let us consider more closely the steps taken by the client remote login process. As in the server process, the first step is to locate the service definition for a remote login:

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

Next the destination host is looked up with a *gethostbyname* call:

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

With this accomplished, all that is required is to establish a connection to the server at the requested host and start up the remote login protocol. The address buffer is cleared, then filled in with the Internet address of the foreign host and the port number at which the login process resides on the foreign host:

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

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

```
s = socket(hp->h_addrtype, SOCK_STREAM, 0);
if (s < 0) {
        perror("rlogin: socket");
        exit(3);
}
 ...
if (connect(s, (struct sockaddr *) &server, sizeof (server)) < 0) {
        perror("rlogin: connect");
        exit(4);
}
```

The details of the remote login protocol will not be considered here.

4.3. Connectionless servers

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

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

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

Note that the use of broadcast for such a task is fairly inefficient, as all hosts must process each message, whether or not using an rwho server. Unless such a service is sufficiently universal and is frequently used, the expense of periodic broadcasts outweighs the simplicity.

The rwho server, in a simplified form, is pictured in Figure 4. There are two separate tasks performed by the server. The first task is to act as a receiver of status information broadcast by other hosts on the network. This job is carried out in the main loop of the program. Packets received at the rwho port are

```
arpa       up     9:45,        5 users, load   1.15,   1.39,   1.31
cad        up     2+12:04,     8 users, load   4.67,   5.13,   4.59
calder     up     10:10,       0 users, load   0.27,   0.15,   0.14
dali       up     2+06:28,     9 users, load   1.04,   1.20,   1.65
degas      up     25+09:48,    0 users, load   1.49,   1.43,   1.41
ear        up     5+00:05,     0 users, load   1.51,   1.54,   1.56
ernie      down   0:24
esvax      down   17:04
ingres     down   0:26
kim        up     3+09:16,     8 users, load   2.03,   2.46,   3.11
matisse    up     3+06:18,     0 users, load   0.03,   0.03,   0.05
medea      up     3+09:39,     2 users, load   0.35,   0.37,   0.50
merlin     down   19+15:37
miro       up     1+07:20,     7 users, load   4.59,   3.28,   2.12
monet      up     1+00:43,     2 users, load   0.22,   0.09,   0.07
oz         down   16:09
statvax    up     2+15:57,     3 users, load   1.52,   1.81,   1.86
ucbvax     up     9:34,        2 users, load   6.08,   5.16,   3.28
```

Figure 3. ruptime output.

interrogated to insure they've been sent by another rwho server process, then are time stamped with their arrival time and used to update a file indicating the status of the host. When a host has not been heard from for an extended period of time, the database interpretation routines assume the host is down and indicate such on the status reports. This algorithm is prone to error as a server may be down while a host is actually up, but serves our current needs.

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

Status information must be broadcast on the local network. For networks which do not support the notion of broadcast another scheme must be used to simulate or replace broadcasting. One possibility is to enumerate the known neighbors (based on the status messages received from other rwho servers). This, unfortunately, requires some bootstrapping information, for a server will have no idea what machines are its neighbors until it receives status messages from them. Therefore, if all machines on a net are freshly booted, no machine will have any known neighbors and thus never receive, or send, any status information. This is the identical problem faced by the routing table management process in propagating routing status information. The standard solution, unsatisfactory as it may be, is to inform one or more servers of known neighbors and request that they always communicate with these neighbors. If each server has at least one neighbor supplied to it, status information may then propagate through a neighbor to hosts which are not (possibly) directly neighbors. If the server is able to support networks which provide a broadcast capability, as well as those which do not, then networks with an arbitrary topology may share status information*.

It is important that software operating in a distributed environment not have any site-dependent information compiled into it. This would require a separate copy of the server at each host and make maintenance a severe headache. 4.3BSD attempts to isolate host-specific information from applications by providing system calls which return the necessary information*. A mechanism exists, in the form of an

* One must, however, be concerned about "loops". That is, if a host is connected to multiple networks, it will receive status information from itself. This can lead to an endless, wasteful, exchange of information.

* An example of such a system call is the *gethostname*(2) call which returns the host's "official" name.

```
main()
{
        ...
        sp = getservbyname("who", "udp");
        net = getnetbyname("localnet");
        sin.sin_addr = inet_makeaddr(INADDR_ANY, net);
        sin.sin_port = sp->s_port;
        ...
        s = socket(AF_INET, SOCK_DGRAM, 0);
        ...
        on = 1;
        if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on)) < 0) {
                syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
                exit(1);
        }
        bind(s, (struct sockaddr *) &sin, sizeof (sin));
        ...
        signal(SIGALRM, onalrm);
        onalrm();
        for (;;) {
                struct whod wd;
                int cc, whod, len = sizeof (from);

                cc = recvfrom(s, (char *)&wd, sizeof (struct whod), 0,
                   (struct sockaddr *)&from, &len);
                if (cc <= 0) {
                        if (cc < 0 && errno != EINTR)
                                syslog(LOG_ERR, "rwhod: recv: %m");
                        continue;
                }
                if (from.sin_port != sp->s_port) {
                        syslog(LOG_ERR, "rwhod: %d: bad from port",
                                ntohs(from.sin_port));
                        continue;
                }
                ...
                if (!verify(wd.wd_hostname)) {
                        syslog(LOG_ERR, "rwhod: malformed host name from %x",
                                ntohl(from.sin_addr.s_addr));
                        continue;
                }
                (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
                whod = open(path, O_WRONLY | O_CREAT | O_TRUNC, 0666);
                ...
                (void) time(&wd.wd_recvtime);
                (void) write(whod, (char *)&wd, cc);
                (void) close(whod);
        }
}
```

Figure 4. rwho server.

ioctl call, for finding the collection of networks to which a host is directly connected. Further, a local network broadcasting mechanism has been implemented at the socket level. Combining these two features allows a process to broadcast on any directly connected local network which supports the notion of broadcasting in a site independent manner. This allows 4.3BSD to solve the problem of deciding how to propagate status information in the case of *rwho*, or more generally in broadcasting: Such status information is broadcast tc connected networks at the socket level, where the connected networks have been obtained via the appropriate *ioctl* calls. The specifics of such broadcastings are complex, however, and will be covered in section 5.

5. ADVANCED TOPICS

A number of facilities have yet to be discussed. For most users of the IPC the mechanisms already described will suffice in constructing distributed applications. However, others will find the need to utilize some of the features which we consider in this section.

5.1. Out of band data

The stream socket abstraction includes the notion of "out of band" data. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data. The abstraction defines that the out of band data facilities must support the reliable delivery of at least one out of band message at a time. This message may contain at least one byte of data, and at least one message may be pending delivery to the user at any one time. For communications protocols which support only in-band signaling (i.e. the urgent data is delivered in sequence with the normal data), the system normally extracts the data from the normal data stream and stores it separately. This allows users to choose between receiving the urgent data in order and receiving it out of sequence without having to buffer all the intervening data. It is possible to "peek" (via MSG_PEEK) at out of band data. If the socket has a process group, a SIGURG signal is generated when the protocol is notified of its existence. A process can set the process group or process id to be informed by the SIGURG signal via the appropriate *fcntl* call, as described below for SIGIO. If multiple sockets may have out of band data awaiting delivery, a *select* call for exceptional conditions may be used to determine those sockets with such data pending. Neither the signal nor the select indicate the actual arrival of the out-of-band data, but only notification that it is pending.

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

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

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

If *yes* is a 1 on return, the next read will return data after the mark. Otherwise (assuming out of band data has arrived), the next read will provide data sent by the client prior to transmission of the out of band signal. The routine used in the remote login process to flush output on receipt of an interrupt or quit signal is shown in Figure 5. It reads the normal data up to the mark (to discard it), then reads the out-of-band byte.

A process may also read or peek at the out-of-band data without first reading up to the mark. This is more difficult when the underlying protocol delivers the urgent data in-band with the normal data, and only sends notification of its presence ahead of time (e.g., the TCP protocol used to implement streams in the Internet domain). With such protocols, the out-of-band byte may not yet have arrived when a *recv* is done with the MSG_OOB flag. In that case, the call will return an error of EWOULDBLOCK. Worse, there may be enough in-band data in the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data that the urgent data may be delivered.

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

```
#include <sys/ioctl.h>
#include <sys/file.h>
 ...
oob()
{
        int out = FWRITE;
        char waste[BUFSIZ], mark;

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

Figure 5. Flushing terminal I/O on receipt of out of band data.

5.2. Non-Blocking Sockets

It is occasionally convenient to make use of sockets which do not block; that is, I/O requests which cannot complete immediately and would therefore cause the process to be suspended awaiting completion are not executed, and an error code is returned. Once a socket has been created via the *socket* call, it may be marked as non-blocking by *fcntl* as follows:

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

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

5.3. Interrupt driven socket I/O

The SIGIO signal allows a process to be notified via a signal when a socket (or more generally, a file descriptor) has data waiting to be read. Use of the SIGIO facility requires three steps: First, the process must set up a SIGIO signal handler by use of the *signal* or *sigvec* calls. Second, it must set the process id or process group id which is to receive notification of pending input to its own process id, or the process group id of its process group (note that the default process group of a socket is group zero). This is accomplished by use of an *fcntl* call. Third, it must enable asynchronous notification of pending I/O requests with another *fcntl* call. Sample code to allow a given process to receive information on pending I/O requests as they occur for a socket *s* is given in Figure 6. With the addition of a handler for SIGURG, this code can also be used to prepare for receipt of SIGURG signals.

```
#include <fcntl.h>
 ...
int     io_handler();
 ...
signal(SIGIO, io_handler);

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

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

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

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

Figure 6. Use of asynchronous notification of I/O requests.

5.4. Signals and process groups

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

Another signal which is useful when constructing server processes is SIGCHLD. This signal is delivered to a process when any child processes have changed state. Normally servers use the signal to ''reap'' child processes that have exited without explicitly awaiting their termination or periodic polling for exit status. For example, the remote login server loop shown in Figure 2 may be augmented as shown in Figure 7.

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

5.5. Pseudo terminals

Many programs will not function properly without a terminal for standard input and output. Since sockets do not provide the semantics of terminals, it is often necessary to have a process communicating over the network do so through a *pseudo-terminal*. A pseudo- terminal is actually a pair of devices, master

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

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

        while (wait3(&status, WNOHANG, 0) > 0)
                ;
}
```

Figure 7. Use of the SIGCHLD signal.

and slave, which allow a process to serve as an active agent in communication between processes and users. Data written on the slave side of a pseudo-terminal is supplied as input to a process reading from the master side, while data written on the master side are processed as terminal input for the slave. In this way, the process manipulating the master side of the pseudo-terminal has control over the information read and written on the slave side as if it were manipulating the keyboard and reading the screen on a real terminal. The purpose of this abstraction is to preserve terminal semantics over a network connection— that is, the slave side appears as a normal terminal to any process reading from or writing to it.

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

Under 4.3BSD, the name of the slave side of a pseudo-terminal is of the form */dev/ttyxy*, where *x* is a single letter starting at 'p' and continuing to 't'. *y* is a hexadecimal digit (i.e., a single character in the range 0 through 9 or 'a' through 'f'). The master side of a pseudo-terminal is */dev/ptyxy*, where *x* and *y* correspond to the slave side of the pseudo-terminal.

In general, the method of obtaining a pair of master and slave pseudo-terminals is to find a pseudo-terminal which is not currently in use. The master half of a pseudo-terminal is a single-open device; thus, each master may be opened in turn until an open succeeds. The slave side of the pseudo-terminal is then opened, and is set to the proper terminal modes if necessary. The process then *fork*s; the child closes the master side of the pseudo-terminal, and *exec*s the appropriate program. Meanwhile, the parent closes the slave side of the pseudo-terminal and begins reading and writing from the master side. Sample code making use of pseudo-terminals is given in Figure 8; this code assumes that a connection on a socket *s* exists, connected to a peer who wants a service of some kind, and that the process has disassociated itself from any previous controlling terminal.

```
gotpty = 0;
for (c = 'p'; !gotpty && c <= 's'; c++) {
        line = "/dev/ptyXX";
        line[sizeof("/dev/pty")-1] = c;
        line[sizeof("/dev/ptyp")-1] = '0';
        if (stat(line, &statbuf) < 0)
                break;
        for (i = 0; i < 16; i++) {
                line[sizeof("/dev/ptyp")-1] = "0123456789abcdef"[i];
                master = open(line, O_RDWR);
                if (master > 0) {
                        gotpty = 1;
                        break;
                }
        }
}
if (!gotpty) {
        syslog(LOG_ERR, "All network ports in use");
        exit(1);
}

line[sizeof("/dev/")-1] = 't';
slave = open(line, O_RDWR); /* slave is now slave side */
if (slave < 0) {
        syslog(LOG_ERR, "Cannot open slave pty %s", line);
        exit(1);
}

ioctl(slave, TIOCGETP, &b);  /* Set slave tty modes */
b.sg_flags = CRMOD|XTABS|ANYP;
ioctl(slave, TIOCSETP, &b);

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

Figure 8. Creation and use of a pseudo terminal

5.6. Selecting specific protocols

If the third argument to the *socket* call is 0, *socket* will select a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. However, when using "raw" sockets to communicate directly with lower-level protocols or hardware interfaces, the protocol argument may be important for setting up demultiplexing. For example, raw sockets in the Internet family may be used to implement a new protocol above IP, and the socket will receive packets only for the protocol specified. To obtain a particular protocol one determines the protocol number as defined within the communication domain. For the Internet domain one may use one of the library routines discussed in section 3, such as *getprotobyname*:

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

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

In the NS domain, the available socket protocols are defined in *<netns/ns.h>*. To create a raw socket for Xerox Error Protocol messages, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
 ...
s = socket(AF_NS, SOCK_RAW, NSPROTO_ERROR);
```

5.7. Address binding

As was mentioned in section 2, binding addresses to sockets in the Internet and NS domains can be fairly complex. As a brief reminder, these associations are composed of local and foreign addresses, and local and foreign ports. Port numbers are allocated out of separate spaces, one for each system and one for each domain on that system. Through the *bind* system call, a process may specify half of an association, the <local address, local port> part, while the *connect* and *accept* primitives are used to complete a socket's association by specifying the <foreign address, foreign port> part. Since the association is created in two steps the association uniqueness requirement indicated previously could be violated unless care is taken. Further, it is unrealistic to expect user programs to always know proper values to use for the local address and local port since a host may reside on multiple networks and the set of allocated port numbers is not directly accessible to a user.

To simplify local address binding in the Internet domain the notion of a "wildcard" address has been provided. When an address is specified as INADDR_ANY (a manifest constant defined in <netinet/in.h>), the system interprets the address as "any valid address". For example, to bind a specific port number to a socket, but leave the local address unspecified, the following code might be used:

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

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

In a similar fashion, a local port may be left unspecified (specified as zero), in which case the system will select an appropriate port number for it. This shortcut will work both in the Internet and NS domains. For example, to bind a specific local address to a socket, but to leave the local port number unspecified:

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

The system selects the local port number based on two criteria. The first is that on 4BSD systems, Internet ports below IPPORT_RESERVED (1024) (for the Xerox domain, 0 through 3000) are reserved for privileged users (i.e., the super user); Internet ports above IPPORT_USERRESERVED (50000) are reserved for non-privileged servers. The second is that the port number is not currently bound to some other socket. In order to find a free Internet port number in the privileged range the *rresvport* library routine may be used as follows to return a stream socket in with a privileged port number:

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

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

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

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

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

5.8. Broadcasting and determining network configuration

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

To send a broadcast message, a datagram socket should be created:

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

or

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

The socket is marked as allowing broadcasting,

```
int    on = 1;

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

and at least a port number should be bound to the socket:

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

or, for the NS domain,

```
sns.sns_family = AF_NS;
netnum = htonl(net);
sns.sns_addr.x_net = *(union ns_net *) &netnum; /* insert net number */
sns.sns_addr.x_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sns, sizeof (sns));
```

The destination address of the message to be broadcast depends on the network(s) on which the message is to be broadcast. The Internet domain supports a shorthand notation for broadcast on the local network, the address INADDR_BROADCAST (defined in <*netinet/in.h*>. To determine the list of addresses for all reachable neighbors requires knowledge of the networks to which the host is connected. Since this information should be obtained in a host-independent fashion and may be impossible to derive, 4.3BSD provides a method of retrieving this information from the system data structures. The SIOCGIFCONF *ioctl* call returns the interface configuration of a host in the form of a single *ifconf* structure; this structure contains a "data area" which is made up of an array of of *ifreq* structures, one for each network interface to which the host is connected. These structures are defined in *<net/if.h>* as follows:

```
struct ifconf {
        int       ifc_len;                          /* size of associated buffer */
        union {
                caddr_t ifcu_buf;
                struct   ifreq *ifcu_req;
        } ifc_ifcu;
};

#define ifc_buf ifc_ifcu.ifcu_buf                   /* buffer address */
#define ifc_req ifc_ifcu.ifcu_req                   /* array of structures returned */

#define IFNAMSIZ       16

struct ifreq {
        char      ifr_name[IFNAMSIZ];               /* if name, e.g. "en0" */
        union {
                struct   sockaddr ifru_addr;
                struct   sockaddr ifru_dstaddr;
                struct   sockaddr ifru_broadaddr;
                short    ifru_flags;
                caddr_t  ifru_data;
        } ifr_ifru;
};

#define ifr_addr        ifr_ifru.ifru_addr        /* address */
#define ifr_dstaddr     ifr_ifru.ifru_dstaddr     /* other end of p-to-p link */
#define ifr_broadaddr   ifr_ifru.ifru_broadaddr   /* broadcast address */
#define ifr_flags       ifr_ifru.ifru_flags       /* flags */
#define ifr_data        ifr_ifru.ifru_data        /* for use by interface */
```

The actual call which obtains the interface configuration is

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

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

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

For each structure there exists a set of "interface flags" which tell whether the network corresponding to that interface is up or down, point to point or broadcast, etc. The SIOCGIFFLAGS *ioctl* retrieves these flags for an interface specified by an *ifreq* structure as follows:

```
struct ifreq *ifr;

ifr = ifc.ifc_req;

for (n = ifc.ifc_len / sizeof (struct ifreq); --n >= 0; ifr++) {
        /*
         * We must be careful that we don't use an interface
         * devoted to an address family other than those intended;
         * if we were interested in NS interfaces, the
         * AF_INET would be AF_NS.
         */
        if (ifr->ifr_addr.sa_family != AF_INET)
                continue;
        if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
                ...
        }
        /*
         * Skip boring cases.
         */
        if ((ifr->ifr_flags & IFF_UP) == 0 ||
           (ifr->ifr_flags & IFF_LOOPBACK) ||
           (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTTOPOINT)) == 0)
                continue;
```

Once the flags have been obtained, the broadcast address must be obtained. In the case of broadcast networks this is done via the SIOCGIFBRDADDR *ioctl*, while for point-to-point networks the address of the destination host is obtained with SIOCGIFDSTADDR.

```
struct sockaddr dst;

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

After the appropriate *ioctl*'s have obtained the broadcast or destination address (now in *dst*), the *sendto* call may be used:

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

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

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

5.9. Socket Options

It is possible to set and get a number of options on sockets via the *setsockopt* and *getsockopt* system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

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

and

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

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

An example should help clarify things. It is sometimes useful to determine the type (e.g., stream, datagram, etc.) of an existing socket; programs under *inetd* (described below) may need to perform this task. This can be accomplished as follows via the SO_TYPE socket option and the *getsockopt* call:

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

int type, size;

size = sizeof (int);

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

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

5.10. NS Packet Sequences

The semantics of NS connections demand that the user both be able to look inside the network header associated with any incoming packet and be able to specify what should go in certain fields of an outgoing packet. Using different calls to *setsockopt*, it is possible to indicate whether prototype headers will be associated by the user with each outgoing packet (SO_HEADERS_ON_OUTPUT), to indicate whether the headers received by the system should be delivered to the user (SO_HEADERS_ON_INPUT), or to indicate default information that should be associated with all outgoing packets on a given socket (SO_DEFAULT_HEADERS).

The contents of a SPP header (minus the IDP header) are:

```
struct sphdr {
        u_char  sp_cc;                  /* connection control */
#define SP_SP  0x80                     /* system packet */
#define SP_SA  0x40                     /* send acknowledgement */
#define SP_OB  0x20                     /* attention (out of band data) */
#define SP_EM  0x10                     /* end of message */
        u_char  sp_dt;                  /* datastream type */
        u_short sp_sid;                 /* source connection identifier */
        u_short sp_did;                 /* destination connection identifier */
        u_short sp_seq;                 /* sequence number */
        u_short sp_ack;                 /* acknowledge number */
        u_short sp_alo;                 /* allocation number */
};
```

Here, the items of interest are the *datastream type* and the *connection control* fields. The semantics of the datastream type are defined by the application(s) in question; the value of this field is, by default, zero, but it can be used to indicate things such as Xerox's Bulk Data Transfer Protocol (in which case it is set to one). The connection control field is a mask of the flags defined just below it. The user may set or clear the end-of-message bit to indicate that a given message is the last of a given substream type, or may set/clear the attention bit as an alternate way to indicate that a packet should be sent out-of-band. As an example, to associate prototype headers with outgoing SPP packets, consider:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr_ns sns, to;
int s, on = 1;
struct databuf {
      struct sphdr proto_spp;   /* prototype header */
      char buf[534];            /* max. possible data by Xerox std. */
} buf;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &on, sizeof(on));
 ...
buf.proto_spp.sp_dt = 1; /* bulk data */
buf.proto_spp.sp_cc = SP_EM;        /* end-of-message */
strcpy(buf.buf, "hello world\n");
sendto(s, (char *) &buf, sizeof(struct sphdr) + strlen("hello world\n"),
   (struct sockaddr *) &to, sizeof(to));
 ...
```

Note that one must be careful when writing headers; if the prototype header is not written with the data with which it is to be associated, the kernel will treat the first few bytes of the data as the header, with unpredictable results. To turn off the above association, and to indicate that packet headers received by the system should be passed up to the user, one might use:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
struct sockaddr sns;
int s, on = 1, off = 0;
 ...
s = socket(AF_NS, SOCK_SEQPACKET, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_OUTPUT, &off, sizeof(off));
setsockopt(s, NSPROTO_SPP, SO_HEADERS_ON_INPUT, &on, sizeof(on));
 ...
```

Output is handled somewhat differently in the IDP world. The header of an IDP-level packet looks like:

```
struct idp {
        u_short         idp_sum;                /* Checksum */
        u_short         idp_len;                /* Length, in bytes, including header */
        u_char          idp_tc;                 /* Transport Control (i.e., hop count) */
        u_char          idp_pt;                 /* Packet Type (i.e., level 2 protocol) */
        struct ns_addr  idp_dna;                /* Destination Network Address */
        struct ns_addr  idp_sna;                /* Source Network Address */
};
```

The primary field of interest in an IDP header is the *packet type* field. The standard values for this field are (as defined in <*netns/ns.h*>):

```
#define NSPROTO_RI      1       /* Routing Information */
#define NSPROTO_ECHO    2       /* Echo Protocol */
#define NSPROTO_ERROR   3       /* Error Protocol */
#define NSPROTO_PE      4       /* Packet Exchange */
#define NSPROTO_SPP     5       /* Sequenced Packet */
```

For SPP connections, the contents of this field are automatically set to NSPROTO_SPP; for IDP packets, this value defaults to zero, which means ''unknown''.

Setting the value of that field with SO_DEFAULT_HEADERS is easy:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/idp.h>
 ...
struct sockaddr sns;
struct idp proto_idp;           /* prototype header */
int s, on = 1;
 ...
s = socket(AF_NS, SOCK_DGRAM, 0);
 ...
bind(s, (struct sockaddr *) &sns, sizeof (sns));
proto_idp.idp_pt = NSPROTO_PE;  /* packet exchange */
setsockopt(s, NSPROTO_IDP, SO_DEFAULT_HEADERS, (char *) &proto_idp,
   sizeof(proto_idp));
 ...
```

Using SO_HEADERS_ON_OUTPUT is somewhat more difficult. When SO_HEADERS_ON_OUTPUT is turned on for an IDP socket, the socket becomes (for all intents and purposes) a raw socket. In this case, all the fields of the prototype header (except the length and checksum fields, which are computed by the kernel) must be filled in correctly in order for the socket to send and receive data in a sensible manner. To be more specific, the source address must be set to that of the host sending the data; the destination address must be set to that of the host for whom the data is intended; the packet type must be set to whatever value is desired; and the hopcount must be set to some reasonable value (almost always zero). It should also be noted that simply sending data using *write* will not work unless a *connect* or *sendto* call is used, in spite of the fact that it is the destination address in the prototype header that is used, not the one given in either of those calls. For almost all IDP applications , using SO_DEFAULT_HEADERS is easier and more desirable than writing headers.

5.11. Three-way Handshake

The semantics of SPP connections indicates that a three-way handshake, involving changes in the datastream type, should — but is not absolutely required to — take place before a SPP connection is closed. Almost all SPP connections are ''well-behaved'' in this manner; when communicating with any process, it is best to assume that the three-way handshake is required unless it is known for certain that it is not required. In a three-way close, the closing process indicates that it wishes to close the connection by sending a zero-length packet with end-of-message set and with datastream type 254. The other side of the connection indicates that it is OK to close by sending a zero-length packet with end-of-message set and datastream type 255. Finally, the closing process replies with a zero-length packet with substream type 255; at this point, the connection is considered closed. The following code fragments are simplified examples of how one might handle this three-way handshake at the user level; in the future, support for this type of close will probably be provided as part of the C library or as part of the kernel. The first code fragment below illustrates how a process might handle three-way handshake if it sees that the process it is communicating with wants to close the connection:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
read(s, buf, BUFSIZE);
if (((struct sphdr *)buf)->sp_dt == SPPSST_END) {
        /*
         * SPPSST_END indicates that the other side wants to
         * close.
         */
        proto_sp.sp_dt = SPPSST_ENDREPLY;
        proto_sp.sp_cc = SP_EM;
        setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
           sizeof(proto_sp));
        write(s, buf, 0);
        /*
         * Write a zero-length packet with datastream type = SPPSST_ENDREPLY
         * to indicate that the close is OK with us. The packet that we
         * don't see (because we don't look for it) is another packet
         * from the other side of the connection, with SPPSST_ENDREPLY
         * on it it, too. Once that packet is sent, the connection is
         * considered closed; note that we really ought to retransmit
         * the close for some time if we do not get a reply.
         */
        close(s);
}
 ...
```

To indicate to another process that we would like to close the connection, the following code would suffice:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netns/ns.h>
#include <netns/sp.h>
 ...
#ifndef SPPSST_END
#define SPPSST_END 254
#define SPPSST_ENDREPLY 255
#endif
struct sphdr proto_sp;
int s;
 ...
proto_sp.sp_dt = SPPSST_END;
proto_sp.sp_cc = SP_EM;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
   sizeof(proto_sp));
write(s, buf, 0);    /* send the end request */
proto_sp.sp_dt = SPPSST_ENDREPLY;
setsockopt(s, NSPROTO_SPP, SO_DEFAULT_HEADERS, (char *)&proto_sp,
   sizeof(proto_sp));
/*
 * We assume (perhaps unwisely)
 * that the other side will send the
 * ENDREPLY, so we'll just send our final ENDREPLY
 * as if we'd seen theirs already.
 */
write(s, buf, 0);
close(s);
 ...
```

5.12. Packet Exchange

The Xerox standard protocols include a protocol that is both reliable and datagram-oriented. This protocol is known as Packet Exchange (PEX or PE) and, like SPP, is layered on top of IDP. PEX is important for a number of things: Courier remote procedure calls may be expedited through the use of PEX, and many Xerox servers are located by doing a PEX ''BroadcastForServers'' operation. Although there is no implementation of PEX in the kernel, it may be simulated at the user level with some clever coding and the use of one peculiar *getsockopt*. A PEX packet looks like:

```
/*
 * The packet-exchange header shown here is not defined
 * as part of any of the system include files.
 */
struct pex {
        struct idp  p_idp;                  /* idp header */
        u_short     ph_id[2];               /* unique transaction ID for pex */
        u_short     ph_client;              /* client type field for pex */
};
```

The *ph_id* field is used to hold a ''unique id'' that is used in duplicate suppression; the *ph_client* field indicates the PEX client type (similar to the packet type field in the IDP header). PEX reliability stems from the fact that it is an idempotent (''I send a packet to you, you send a packet to me'') protocol. Processes on each side of the connection may use the unique id to determine if they have seen a given packet before (the unique id field differs on each packet sent) so that duplicates may be detected, and to indicate which message a given packet is in response to. If a packet with a given unique id is sent and no response is received in a given amount of time, the packet is retransmitted until it is decided that no response will ever be received. To simulate PEX, one must be able to generate unique ids -- something that is hard to do at the user level with any real guarantee that the id is really unique. Therefore, a means (via *getsockopt*) has been provided for getting unique ids from the kernel. The following code fragment indicates how to get a unique id:

```
long uniqueid;
int s, idsize = sizeof(uniqueid);
...
s = socket(AF_NS, SOCK_DGRAM, 0);
...
/* get id from the kernel -- only on IDP sockets */
getsockopt(s, NSPROTO_PE, SO_SEQNO, (char *)&uniqueid, &idsize);
...
```

The retransmission and duplicate suppression code required to simulate PEX fully is left as an exercise for the reader.

5.13. Inetd

One of the daemons provided with 4.3BSD is *inetd*, the so called ''internet super-server.'' *Inetd* is invoked at boot time, and determines from the file */etc/inetd.conf* the servers for which it is to listen. Once this information has been read and a pristine environment created, *inetd* proceeds to create one socket for each service it is to listen for, binding the appropriate port number to each socket.

Inetd then performs a *select* on all these sockets for read availability, waiting for somebody wishing a connection to the service corresponding to that socket. *Inetd* then performs an *accept* on the socket in question, *forks*, *dups* the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and *execs* the appropriate server.

Servers making use of *inetd* are considerably simplified, as *inetd* takes care of the majority of the IPC work required in establishing a connection. The server invoked by *inetd* expects the socket connected to its client on file descriptors 0 and 1, and may immediately perform any operations such as *read*, *write*, *send*, or *recv*. Indeed, servers may use buffered I/O as provided by the ''stdio'' conventions, as long as as they remember to use *fflush* when appropriate.

One call which may be of interest to individuals writing servers under *inetd* is the *getpeername* call, which returns the address of the peer (process) connected on the other end of the socket. For example, to log the Internet address in ''dot notation'' (e.g., ''128.32.0.4'') of a client connected to a server under *inetd*, the following code might be used:

```
struct sockaddr_in name;
int namelen = sizeof (name);
...
if (getpeername(0, (struct sockaddr *)&name, &namelen) < 0) {
      syslog(LOG_ERR, "getpeername: %m");
      exit(1);
} else
      syslog(LOG_INFO, "Connection from %s", inet_ntoa(name.sin_addr));
...
```

While the *getpeername* call is especially useful when writing programs to run with *inetd*, it can be used under other circumstances. Be warned, however, that *getpeername* will fail on UNIX domain sockets.

Printed in the U.S.A.

172-054-001 A1