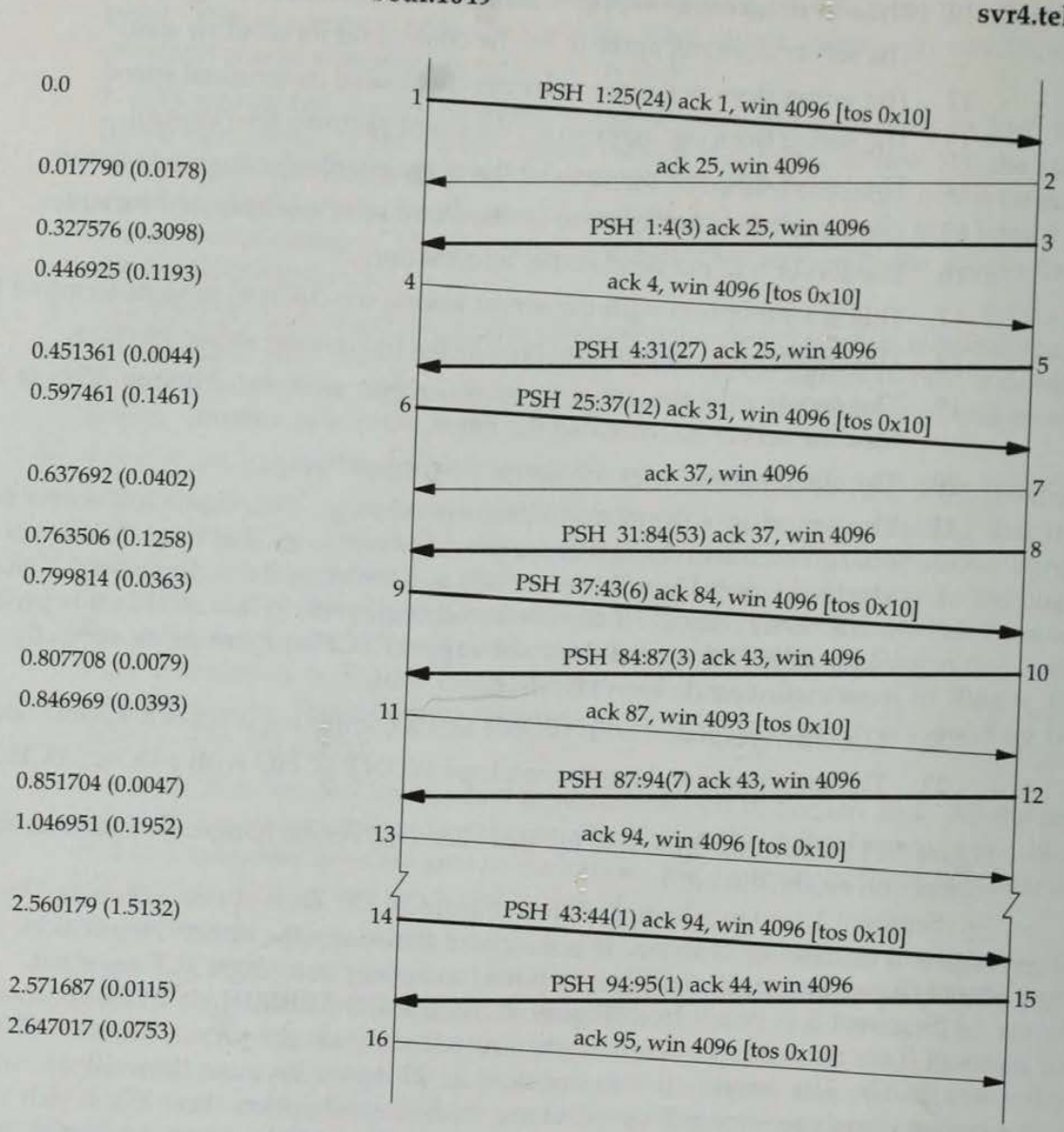


Figure 26.13 Initial option negotiation by Telnet client and server.

Segment 9 contains the final two options from the client: lines 20 and 22. The response in segment 10 is line 23, the final option from the server.

From this point in the time line user data is exchanged across the connection. There is nothing to prevent additional option negotiation, we just don't see any in this example. Segment 12 is the `login:` prompt from the server. Segment 14 is the first character we type of our login name, with its echo returned in segment 15. This is the type of interactive traffic we saw in Section 19.2 with Rlogin: one character at a time sent by the client, with the server performing the echo.

The option negotiation in Figure 26.12 is initiated by the client, but throughout this text we've been using the Telnet client to connect to standard servers such as the daytime server and the echo server, to demonstrate various feature of TCP. When we watched the packet exchange in these examples, such as Figure 18.1, we never saw the client initiate option negotiation. Why? The Unix Telnet client does not initiate any option negotiation if a port number other than the standard Telnet port (23) is specified. This lets the Telnet client, using the standard NVT, exchange data with other, non-Telnet servers. We've used it with the daytime, echo, and discard servers throughout the text, and we'll use it with the FTP and SMTP servers in later chapters.

Linemode

To see Telnet's linemode option in action we'll run the client on our host `bsdi`, connecting to the 4.4BSD server on `vangogh.cs.berkeley.edu`. Both BSD/386 and 4.4BSD support this option.

We won't go through all the packets and option and suboption negotiations, because they're similar to the previous example and the linemode option is quite detailed. Instead we'll note the following differences with the option negotiation.

1. The 4.4BSD server supports more of the options that the BSD/386 tries to negotiate: window size, local flow control, status, accepting environment variables, and terminal speed.
2. The 4.4BSD server tries to negotiate a newer option that the BSD/386 client doesn't support: authentication (to avoid sending the user's password in cleartext across the connection).
3. The client sends the WILL LINEMODE option, as before, but the server responds with DO LINEMODE, since it's supported. This causes the client to send its 16 special characters to the server as a suboption. These are the current values of the special terminal characters in effect at the client: the interrupt character, the end-of-file character, and so on.

   The server sends a suboption to the client telling the client to process all input lines, performing any editing functions (erase character, erase line, etc.). The client sends only completed lines to the server. The server also tells the client to translate any interrupts keys or signal keys into the corresponding Telnet character. For example, if the interrupt key is Control-C, and we type Control-C to interrupt a running process on the server, the client sends the Telnet IP command (<IAC, IP>) to the server.
4. Another difference occurs when we type our password. With Rlogin and character-at-a-time Telnet, the server is responsible for echoing, so when the server reads the password, it doesn't echo the characters. In linemode, however, the client does the echoing. To handle this, the following exchange takes place:
   (a) The server sends WILL ECHO, telling the client that the server will echo.
   (b) The client responds with DO ECHO.
   (c) The server sends the string `Password:` to the client, and the client outputs the string to the terminal.

(d) We type our password and the client sends it to the server when we type the RETURN key. The password is not echoed, since the client thinks the server will echo it.

(e) The server sends the 2-byte sequence CR, LF, to move the cursor, since the RETURN that completed the password was not echoed.

(f) The server sends WONT ECHO.

(g) The client responds with DONT ECHO. The client resumes echoing.

Once we login, the client builds complete lines and sends them to the server. This is the intent of the linemode option. It reduces the number of segments exchanged between the client and server, and provides faster response to client keystrokes (i.e., echoing and editing). Figure 26.14 shows the packet exchange when we type the command

```
vangogh % date
```

across a Telnet connection using linemode. (We have removed the type-of-service information, along with the window advertisements.)

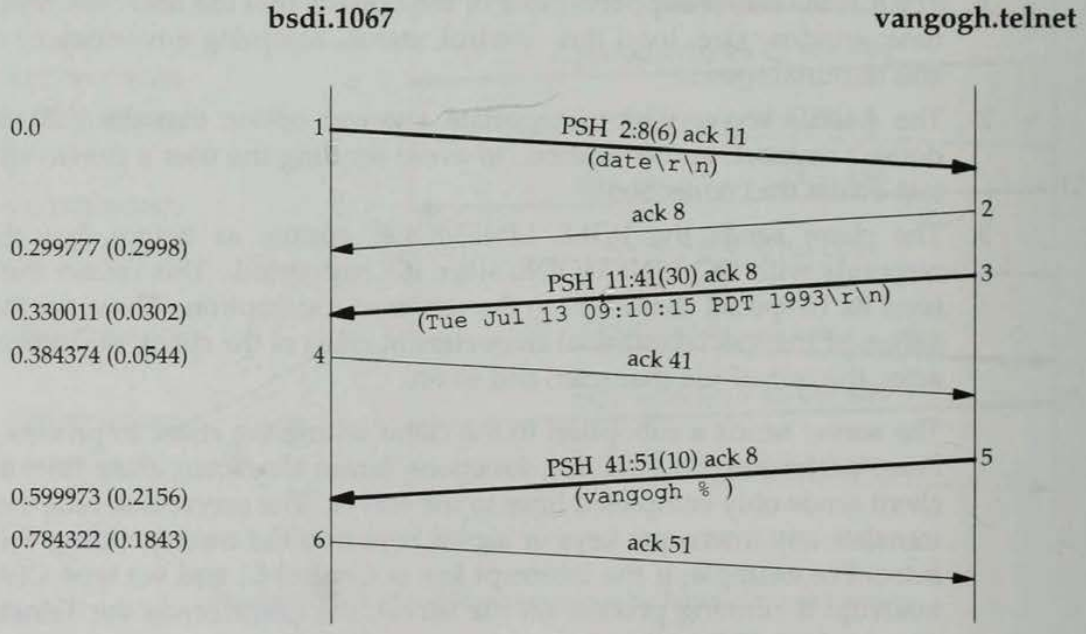


Figure 26.14 Sending a command from client to server using Telnet linemode.

If we compare this with the same command typed to Rlogin (Figure 19.2) we see that Telnet's linemode uses two segments (one with data, one for the ACK, total of 86 bytes including the IP and TCP headers), while 15 segments are used with Rlogin (5 with typed data, 5 with echoed data, 5 ACKs, total of 611 bytes). That is an enormous savings!

What if we run an application on the server that needs to use the single-character mode? (The `vi` editor is an example.) The following steps take place.

1. When the application starts on the server and changes the mode of its pseudo-terminal, the Telnet server is notified that single-character mode is required. The server sends WILL ECHO to the client, along with the linemode suboption that tells the client not to build complete lines, but to send one character at a time.
2. The client responds with DO ECHO and acknowledges the linemode suboption.
3. The application runs on the server. Each character we type is sent to the server by itself (constrained by the Nagle algorithm, of course), and the server does any required echoing.
4. When the application terminates, and restores the mode of its pseudo-terminal, the Telnet server is notified. The server sends WONT ECHO to the client, along with the linemode suboption telling the client to build complete lines.
5. The client responds with DONT ECHO and acknowledges the linemode suboption.

The difference between this scenario and typing our password shows that the echo function and character-at-a-time versus line-at-a-time are independent features. When we type our password, echo must be off but line-at-a-time is OK. For a full-screen application such as an editor, echo is off and character-at-a-time is required.

Figure 26.15 summarizes the different modes we've seen with Telnet and Rlogin.

Application	Client sends		Client echo?	Example
	character at-a-time	line at-a-time		
Rlogin	•		no	
Telnet, character at a time	•		no	
Telnet, linemode		•	yes	normal commands
Telnet, linemode		•	no	typing our password
Telnet, linemode	•		no	`vi` editor

Figure 26.15 Comparison of Rlogin and Telnet modes of operation.

Line-at-a-Time Mode (Kludge Line Mode)

We saw in Figure 26.11 that newer servers that support the linemode option go into kludge line-at-a-time mode, if the client doesn't support linemode. We also mentioned that all the clients and servers in that figure supported the kludge line mode, but it wasn't the default, and must be explicitly enabled by the server or by the user. Let's see how the kludge line mode is enabled, using Telnet options.

We first describe how the BSD/386 server negotiates this mode, when the client doesn't support real linemode.

1. When the client rejects the server's request to enable linemode, the server sends the DO TIMING MARK option. RFC 860 [Postel and Reynolds 1983f] defines

this Telnet option. It is intended for the two ends to synchronize with each other, as we'll see later in this section when we look at user interrupts. This use of the option is just to determine if the client supports the kludge line mode.

2. The client responds with a WILL TIMING MARK, indicating it supports the kludge line mode.
3. The server sends the WONT SUPPRESS GO AHEAD option along with the WONT ECHO option, saying that it wants to disable these two options. We mentioned earlier that character-at-a-time mode assumes that both SUPPRESS GO AHEAD and ECHO are both on, so turning off these options is the kludge that starts line mode.
4. The client responds with DONT SUPPRESS GO AHEAD and DONT ECHO.
5. The `login:` prompt is sent by the server and we type our login name. It is sent to the server as a complete line and echoed locally by the client.
6. The server sends the string `Password:` along with the WILL ECHO option. This turns off echoing of the password we type by the Telnet client, because it thinks the server will echo. The client responds with DO ECHO.
7. We type our password. It is sent by the client to the server as a complete line.
8. Echoing is turned back on by the server sending WONT ECHO, which the client responds to with DONT ECHO.

At this point normal commands are handled similar to the linemode option. The client performs all editing and echoing, sending complete lines to the server.

We mentioned earlier that all the clients and servers in Figure 26.11 with an entry of "char" support the kludge line mode, but start by default in character-at-a-time mode. We can easily watch the negotiation that takes place when we tell the client to enter line mode:

```
                                        client is sun, server is svr4
svr4 %                                  type Control-] to talk to Telnet client (not echoed)

telnet> status                          verify currently in character-at-a-time mode
Connected to svr4.tuc.noao.edu
Operating in character-at-a-time mode.
Escape character is '^]'.

telnet> toggle options                  let's watch the option processing
Will show option processing.

telnet> mode line                       and switch to kludge line mode
SENT dont SUPPRESS GO AHEAD             client sends these two options
SENT dont ECHO
RCVD wont SUPPRESS GO AHEAD             and server responds to both with WONT
RCVD wont ECHO
```

This puts the Telnet session in the kludge line mode, with both the SUPPRESS GO AHEAD and ECHO options disabled.

If we run an application such as the `vi` editor on the server, we have the same problem we had with the linemode option. The server needs to tell the client to switch from kludge line mode to character-at-a-time mode while the application runs, and then switch back when it's finished. The following technique is used.

1. The Telnet server knows it must change to character-at-a-time mode because the application changes the mode of its pseudo-terminal, which notifies the server. The server sends WILL SUPPRESS GO AHEAD and WILL ECHO. This puts the client into character-at-a-time mode.
2. The client responds with DO SUPPRESS GO AHEAD and WILL ECHO.
3. The application runs on the server.
4. When the application terminates and changes the mode of its pseudo-terminal, the Telnet server puts the client back into kludge line mode. It sends WONT SUPPRESS GO AHEAD and WONT ECHO.
5. The client responds with DONT SUPPRESS GO AHEAD and DONT ECHO, indicating that it's back in kludge line mode.

Figure 26.16 summarizes the various settings of the SUPPRESS GO AHEAD and ECHO options for character-at-a-time mode and kludge line mode.

Mode	SUPPRESS GO AHEAD	ECHO	Example
character at a time	on	on	`vi` editor during kludge line mode
kludge line mode	off	off	normal commands
kludge line mode	off	on	typing our password

Figure 26.16 Settings of Telnet options during kludge line mode.

Linemode: Client Interrupt Key

Let's see what Telnet does when the client types the interrupt key. We establish a session between the client `bsdi` and the server `vangogh.cs.berkeley.edu`. Figure 26.17 shows the time line when the interrupt key is typed. (We have removed the window advertisements and the type-of-service.)

In segment 1 the interrupt key (often Control-C or DELETE) is converted into Telnet's IP (interrupt process) command: <IAC, IP>. The next 3 bytes, <IAC, DO, TM>, comprise Telnet's DO TIMING MARK option. This mark is sent by the client and must be responded to with either a WILL or a WONT. In either case, all data received from the server before that response is thrown away (except for Telnet commands). This is a synchronization mark from the client to the server and back. Segment 1 is not sent using TCP's urgent mode.

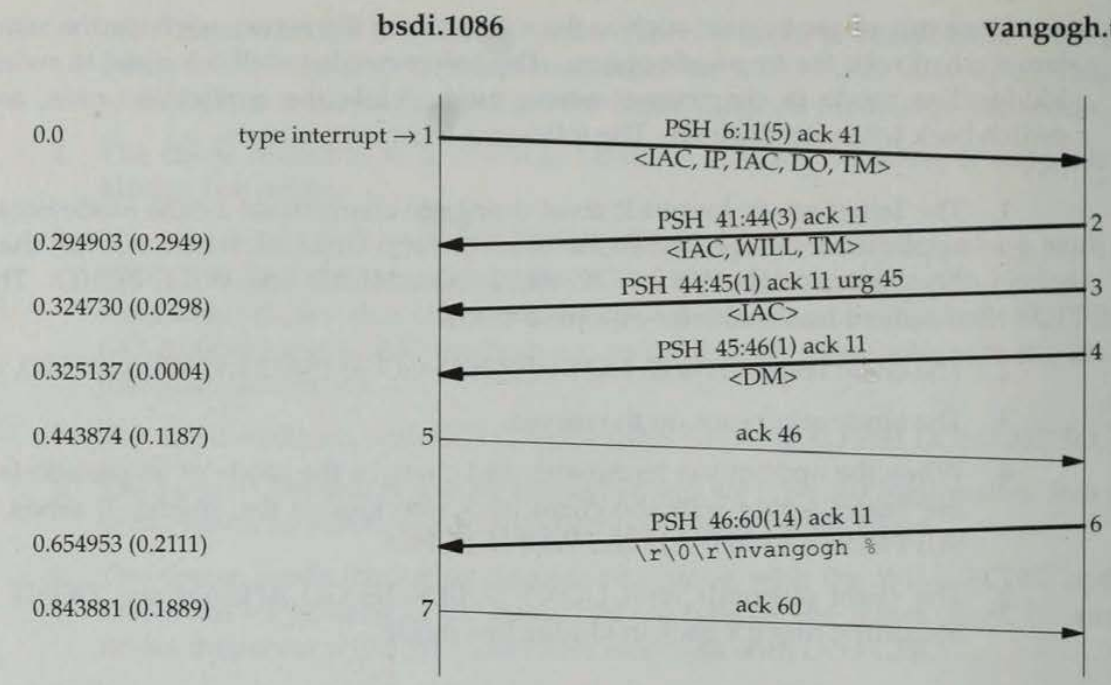


Figure 26.17 Typing interrupt key during linemode operation.

The Host Requirements RFC states that the IP command should be sent using Telnet's synch signal. If it were, the <IAC, IP> would be followed by <IAC, DM>, with the urgent pointer pointing at the DM byte. Most Unix Telnet clients have an option that sends the IP command with the synch signal, but the option defaults off (as we see here).

Segment 2 is the server's reply to the DO TIMING MARK option. It is followed in segments 3 and 4 by a Telnet synch signal: <IAC, DM>. The urgent pointer in segment 3 points to the DM byte, which is sent in segment 4.

If there had been a window full of data queued or in flight from the server to the client, all this data would have been thrown away by the client after sending the IP command in segment 1. Even if the server were stopped by TCP's flow control from sending the data in segments 2, 3, and 4, the urgent pointer is still sent. This is similar to what we saw with Rlogin in Figure 26.7.

Why is the synch signal sent as two segments (3 and 4)? The reason is the problem we detailed in Section 20.8 dealing with TCP's urgent pointer. The Host Requirements RFC says it should point to the last byte of urgent data, while most Berkeley-derived implementations have it point 1 byte beyond the last byte of urgent data. (Recall in Figure 26.6 that the urgent pointer pointed 1 byte beyond the command byte.) The Telnet server purposely writes the first byte of the synch signal as urgent data, knowing the urgent pointer will (incorrectly) point to the next byte that it writes (the data mark, DM), but this first write with the IAC byte is sent immediately, along with the urgent pointer, followed in the next segment by the DM byte.

The final segment of data, segment 6, is just the next shell prompt from the server.

26.6 Summary

This chapter has shown the operation of the Rlogin and Telnet applications. Both provide remote login from a client host to a server host, to let us run programs on the server.

The two applications are different. Rlogin was written assuming both ends of the connection are Unix hosts, so only one option is provided. It is a simple protocol. Telnet, on the other hand, has been around longer and makes no assumptions about the type of host at each end. Telnet is intended to work between different operating systems.

To support a heterogeneous environment, Telnet provides option negotiation between the client and server, to add capabilities if both ends support it. This provides a bare bones implementation for simple clients or servers, but can take advantage of newer features when supported by both ends.

We watched Telnet option negotiation and saw the three types of data transfer: character-at-a-time, kludge line mode, and real linemode. Today the trend is toward line-at-a-time input, when possible, to reduce network traffic and provide better response to the interactive user for line editing and echoing.

Figure 26.18 summarizes and compares the different features provided by Rlogin and Telnet.

Feature	Rlogin	Telnet
transport protocol	One TCP connection. Uses urgent mode.	One TCP connection. Uses urgent mode.
packet mode	Always character-at-a-time, remote echo.	Common default is character-at-a-time, remote echo. Kludge line mode with client echo commonly supported. New option for real linemode with client echo. Always character-at-a-time when application on server requires it.
flow control	Normally done by client, can be disabled by server.	Normally done by server, option allows client to do it.
terminal type	Always provided.	Option, commonly supported.
terminal speed	Always provided.	Option.
window size	Option supported by most servers.	Option.
environment variables	Not supported.	Option.
automatic login	Default. May be prompted for password, which is sent as cleartext. Newer versions support Kerberos login.	Default is to type login name and password. Password is sent as cleartext. Authentication option provided by newer versions.

Figure 26.18 Summary of features provided by Rlogin and Telnet.

Both the Telnet server and the Rlogin server normally set TCP's keepalive option (Chapter 23), if supported by the server's TCP implementation, to detect if the client host crashes. Both applications also use TCP's urgent mode to send server commands to the client even if the flow of data in this direction has been flow control stopped.

Exercises

26.1 Identify all the delayed ACKs in Figure 26.5.

26.2 Why was segment 12 in Figure 26.7 sent?

26.3 We said that the Rlogin client must use a reserved port (Section 1.9). (Normally the Rlogin client only uses reserved ports in the range 512–1023.) What limitation does this present to a host? Is there a way around this?

26.4 Read RFC 1097, describing the Telnet subliminal-message option.

27

FTP: File Transfer Protocol

27.1 Introduction

FTP is another commonly used application. It is the Internet standard for file transfer. We must be careful to differentiate between *file transfer*, which is what FTP provides, and *file access*, which is provided by applications such as NFS (Sun's Network File System, Chapter 29). The file transfer provided by FTP copies a complete file from one system to another system. To use FTP we need an account to login to on the server, or we need to use it with a server that allows *anonymous FTP* (which we show an example of in this chapter).

Like Telnet, FTP was designed from the start to work between different hosts, running different operating systems, using different file structures, and perhaps different character sets. Telnet, however, achieved heterogeneity by forcing both ends to deal with a single standard: the NVT using 7-bit ASCII. FTP handles all the differences between different systems using a different approach. FTP supports a limited number of file types (ASCII, binary, etc.) and file structures (byte stream or record oriented).

RFC 959 [Postel and Reynolds 1985] is the official specification for FTP. This RFC contains a history of the evolution of file transfer over the years.

27.2 FTP Protocol

FTP differs from the other applications that we've described because it uses two TCP connections to transfer a file.

1. The *control connection* is established in the normal client–server fashion. The server does a passive open on the well-known port for FTP (21) and waits for a

client connection. The client does an active open to TCP port 21 to establish the control connection. The control connection stays up for the entire time that the client communicates with this server. This connection is used for commands from the client to the server and for the server's replies.

The IP type-of-service for the control connection should be "minimize delay" since the commands are normally typed by a human user (Figure 3.2).

2. A *data connection* is created each time a file is transferred between the client and server. (It is also created at other times, as we'll see later.)

   The IP type-of-service for the data connection should be "maximize throughput" since this connection is for file transfer.

Figure 27.1 shows the arrangement of the client and server and the two connections between them.

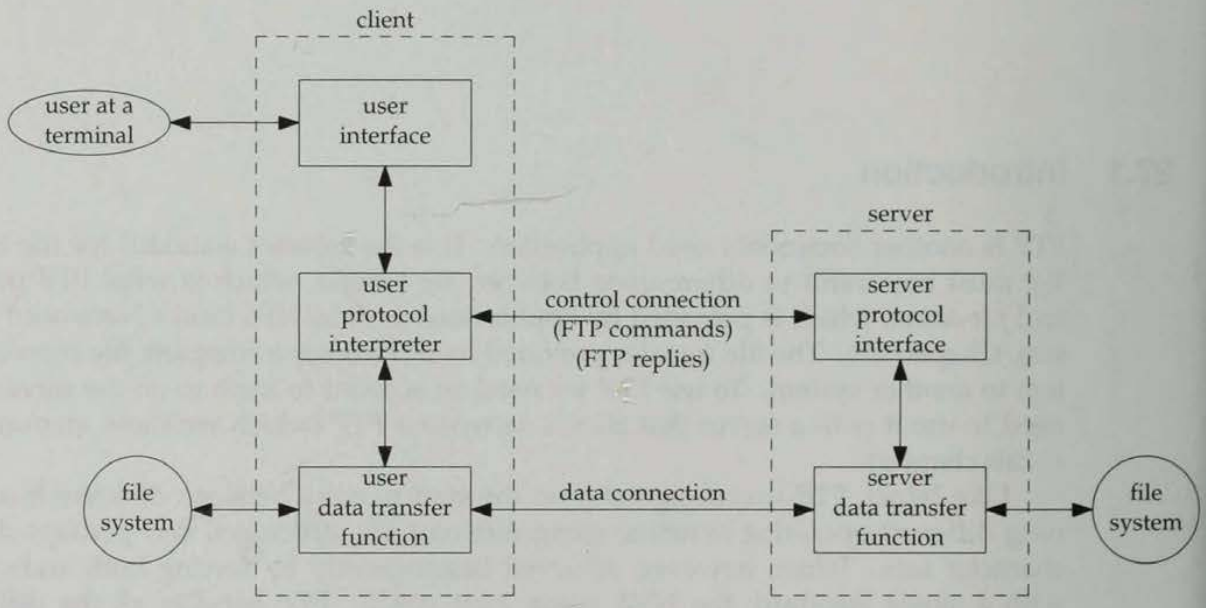


Figure 27.1 Processes involved in file transfer.

This figure shows that the interactive user normally doesn't deal with the commands and replies that are exchanged across the control connection. Those details are left to the two protocol interpreters. The box labeled "user interface" presents whatever type of interface is desired to the interactive user (full-screen menu selection, line-at-a-time commands, etc.) and converts these into FTP commands that are sent across the control connection. Similarly the replies returned by the server across the control connection can be converted to any format to present to the interactive user.

This figure also shows that it is the two protocol interpreters that invoke the two data transfer functions, when necessary.

Data Representation

Numerous choices are provided in the FTP protocol specification to govern the way the file is transferred and stored. A choice must be made in each of four dimensions.

1. File type.
   (a) ASCII file type.
   (Default) The text file is transferred across the data connection in NVT ASCII. This requires the sender to convert the local text file into NVT ASCII, and the receiver to convert NVT ASCII to the local text file type. The end of each line is transferred using the NVT ASCII representation of a carriage return, followed by a linefeed. This means the receiver must scan every byte, looking for the CR, LF pair. (We saw the same scenario with TFTP's ASCII file transfer in Section 15.2.)
   (b) EBCDIC file type.
   An alternative way of transferring text files when both ends are EBCDIC systems.
   (c) Image file type. (Also called binary.)
   The data is sent as a contiguous stream of bits. Normally used to transfer binary files.
   (d) Local file type.
   A way of transferring binary files between hosts with different byte sizes. The number of bits per byte is specified by the sender. For systems using 8-bit bytes, a local file type with a byte size of 8 is equivalent to the image file type.

2. Format control. This choice is available only for ASCII and EBCDIC file types.
   (a) Nonprint.
   (Default) The file contains no vertical format information.
   (b) Telnet format control.
   The file contains Telnet vertical format controls for a printer to interpret.
   (c) Fortran carriage control.
   The first character of each line is the Fortran format control character.

3. Structure.
   (a) File structure.
   (Default) The file is considered as a contiguous stream of bytes. There is no internal file structure.
   (b) Record structure.
   This structure is only used with text files (ASCII or EBCDIC).

(c) Page structure.
Each page is transmitted with a page number to let the receiver store the pages in a random order. Provided by the TOPS-20 operating system. (The Host Requirements RFC recommends against implementing this structure.)

4. Transmission mode. This specifies how the file is transferred across the data connection.
   (a) Stream mode.
   (Default) The file is transferred as a stream of bytes. For a file structure, the end-of-file is indicated by the sender closing the data connection. For a record structure, a special 2-byte sequence indicates the end-of-record and end-of-file.
   (b) Block mode.
   The file is transferred as a series of blocks, each preceded by one or more header bytes.
   (c) Compressed mode.
   A simple run-length encoding compresses consecutive appearances of the same byte. In a text file this would commonly compress strings of blanks, and in a binary file this would commonly compress strings of 0 bytes. (This is rarely used or supported. There are better ways to compress files for FTP.)

If we calculate the number of combinations of all these choices, there could be 72 different ways to transfer and store a file. Fortunately we can ignore many of the options, because they are either antiquated or not supported by most implementations.

Common Unix implementations of the FTP client and server restrict us to the following choices:

- Type: ASCII or image.
- Format control: nonprint only.
- Structure: file structure only.
- Transmission mode: stream mode only.

This limits us to one of two modes: ASCII or image (binary).

> This implementation meets the minimum requirements of the Host Requirements RFC. (This RFC also requires support for the record structure, but only if the operating system supports it, which Unix doesn't.)
>
> Many non-Unix implementations provide FTP capabilities to handle their own file formats. The Host Requirements RFC states "The FTP protocol includes many features, some of which are not commonly implemented. However, for every feature in FTP, there exists at least one implementation."

FTP Commands

The commands and replies sent across the control connection between the client and server are in NVT ASCII. This requires a CR, LF pair at the end of each line (i.e., each command or each reply).

The only Telnet commands (those that begin with IAC) that can be sent by the client to the server are interrupt process (<IAC, IP>) and the Telnet synch signal (<IAC, DM> in urgent mode). We'll see that these two Telnet commands are used to abort a file transfer that is in progress, or to query the server while a transfer is in progress. Additionally, if the server receives a Telnet option command from the client (WILL, WONT, DO, or DONT) it responds with either DONT or WONT.

The commands are 3 or 4 bytes of uppercase ASCII characters, some with optional arguments. More than 30 different FTP commands can be sent by the client to the server. Figure 27.2 shows some of the commonly used commands, most of which we'll encounter in this chapter.

Command	Description
ABOR	abort previous FTP command and any data transfer
LIST *filelist*	list files or directories
PASS *password*	password on server
PORT *n1,n2,n3,n4,n5,n6*	client IP address (*n1.n2.n3.n4*) and port ($n5 \times 256 + n6$)
QUIT	logoff from server
RETR *filename*	retrieve (get) a file
STOR *filename*	store (put) a file
SYST	server returns system type
TYPE *type*	specify file type: A for ASCII, I for image
USER *username*	username on server

Figure 27.2 Common FTP commands.

We'll see in the examples in the next section that sometimes there is a one-to-one correspondence between what the interactive user types and the FTP command sent across the control connection, but for some operations a single user command results in multiple FTP commands across the control connection.

FTP Replies

The replies are 3-digit numbers in ASCII, with an optional message following the number. The intent is that the software needs to look only at the number to determine how to process the reply, and the optional string is for human consumption. Since the clients normally output both the numeric reply and the message string, an interactive user can determine what the reply says by just reading the string (and not have to memorize what all the numeric reply codes mean).

Each of the three digits in the reply code has a different meaning. (We'll see in Chapter 28 that the Simple Mail Transfer Protocol, SMTP, uses the same conventions for commands and replies.)

Figure 27.3 shows the meanings of the first and second digits of the reply code.

Reply	Description
1yz	Positive preliminary reply. The action is being started but expect another reply before sending another command.
2yz	Positive completion reply. A new command can be sent.
3yz	Positive intermediate reply. The command has been accepted but another command must be sent.
4yz	Transient negative completion reply. The requested action did not take place, but the error condition is temporary so the command can be reissued later.
5yz	Permanent negative completion reply. The command was not accepted and should not be retried.
x0z	Syntax errors.
x1z	Information.
x2z	Connections. Replies referring to the control or data connections.
x3z	Authentication and accounting. Replies for the login or accounting commands.
x4z	Unspecified.
x5z	Filesystem status.

Figure 27.3 Meanings of first and second digits of 3-digit reply codes.

The third digit gives additional meaning to the error message. For example, here are some typical replies, along with a possible message string.

- 125 Data connection already open; transfer starting.
- 200 Command OK.
- 214 Help message (for human user).
- 331 Username OK, password required.
- 425 Can't open data connection.
- 452 Error writing file.
- 500 Syntax error (unrecognized command).
- 501 Syntax error (invalid arguments).
- 502 Unimplemented MODE type.

Normally each FTP command generates a one-line reply. For example, the QUIT command could generate the reply:

```
221 Goodbye.
```

If a multiline reply is needed, the first line contains a hyphen instead of a space after the 3-digit reply code, and the final line contains the same 3-digit reply code, followed by a space. For example, the HELP command could generate the reply:

```
214- The following commands are recognized (* =>'s unimplemented).
   USER    PORT    STOR    MSAM*   RNTO    NLST    MKD     CDUP
   PASS    PASV    APPE    MRSQ*   ABOR    SITE    XMKD    XCUP
   ACCT*   TYPE    MLFL*   MRCP*   DELE    SYST    RMD     STOU
   SMNT*   STRU    MAIL*   ALLO    CWD     STAT    XRMD    SIZE
   REIN*   MODE    MSND*   REST    XCWD    HELP    PWD     MDTM
   QUIT    RETR    MSOM*   RNFR    LIST    NOOP    XPWD
214 Direct comments to ftp-bugs@bsdi.tuc.noao.edu.
```

Connection Management

There are three uses for the data connection.

1. Sending a file from the client to the server.
2. Sending a file from the server to the client.
3. Sending a listing of files or directories from the server to the client.

The FTP server sends file listings back across the data connection, rather than as multi-line replies across the control connection. This avoids any line limits that restrict the size of a directory listing and makes it easier for the client to save the output of a directory listing into a file, instead of printing the listing to the terminal.

We've said that the control connection stays up for the duration of the client–server connection, but the data connection can come and go, as required. How are the port numbers chosen for the data connection, and who does the active open and passive open?

First, we said earlier that the common transmission mode (under Unix the *only* transmission mode) is the stream mode, and that the end-of-file is denoted by closing the data connection. This implies that a brand new data connection is required for every file transfer or directory listing. The normal procedure is as follows:

1. The creation of the data connection is under control of the client, because it's the client that issues the command that requires the data connection (get a file, put a file, or list a directory).
2. The client normally chooses an ephemeral port number on the client host for its end of the data connection. The client issues a passive open from this port.
3. The client sends this port number to the server across the control connection using the PORT command.
4. The server receives the port number on the control connection, and issues an active open to that port on the client host. The server's end of the data connection always uses port 20.

Figure 27.4 shows the state of the connections while step 3 is being performed. We assume the client's ephemeral port for the control connection is 1173, and the client's ephemeral port for the data connection is 1174. The command sent by the client is the PORT command and its arguments are six decimal numbers in ASCII, separated by commas. The first four numbers specify the IP address on the client that the server should issue the active open to (140.252.13.34 in this example), and the next two specify the 16-bit port number. Since the 16-bit port number is formed from two numbers, its value in this example is $4 \times 256 + 150 = 1174$.

Figure 27.5 shows the state of the connection when the server issues the active open to the client's end of the data connection. The server's end point is at port 20.

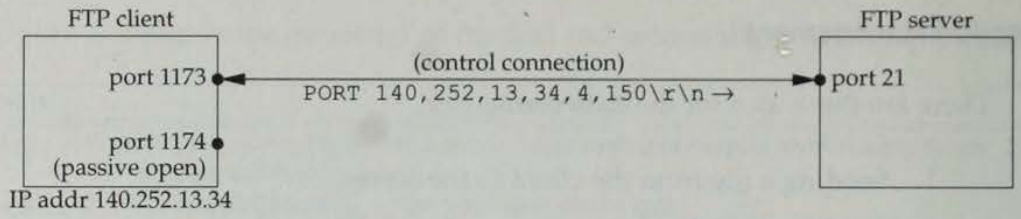


Figure 27.4 PORT command going across FTP control connection.

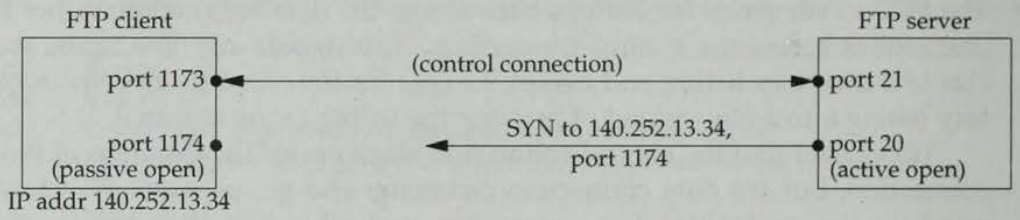


Figure 27.5 FTP server doing active open of data connection.

The server always does the active open of the data connection. Normally the server also does the active close of the data connection, except when the client is sending a file to the server in stream mode, which requires the client to close the connection (which gives the server the end-of-file notification).

It is also possible for the client to not issue the PORT command, in which case the server issues the active open to the same port number being used by the client for its end of the control connection (1173 in this example). This is OK, since the server's port numbers for the two connections are different: one is 20 and the other is 21. Nevertheless, in the next section we'll see why current implementations normally don't do this.

27.3 FTP Examples

We now look at some examples using FTP: its management of the data connection, how text files are sent using NVT ASCII, FTP's use of the Telnet synch signal to abort an in-progress transfer, and finally the popular "anonymous FTP."

Connection Management: Ephemeral Data Port

Let's first look at FTP's connection management with a simple FTP session that just lists a file on the server. We run the client on the host `svr4` with the `-d` flag (debug). This tells it to print the commands and replies that are exchanged across the control connection. All the lines preceded by `--->` are sent by the client to the server, and the lines that begin with a 3-digit number are the server's replies. The client's interactive prompt is `ftp>`.

```
svr4 % ftp -d bsdi                                  -d option for debug output
Connected to bsdi.                                  client does active open of control connection
220 bsdi FTP server (Version 5.60) ready.           server responds it is ready

Name (bsdi:rstevens):                               client prompts us for a login name

---> USER rstevens                                 we type RETURN, so client sends default
331 Password required for rstevens.

Password:                                           we type our password; it's not echoed

---> PASS XXXXXXX                                  client sends it as cleartext
230 User rstevens logged in.

ftp> dir hello.c                                    ask for directory listing of a single file

---> PORT 140,252,13,34,4,150                      see Figure 27.4
200 PORT command successful.

---> LIST hello.c
150 Opening ASCII mode data connection for /bin/ls.

-rw-r--r--  1 rstevens  staff  38 Jul 17 12:47 hello.c

226 Transfer complete.

remote: hello.c                                     output by client
56 bytes received in 0.03 seconds (1.8 Kbytes/s)

ftp> quit                                           we're done

---> QUIT
221 Goodbye.
```

When the FTP client prompts us for a login name, it prints the default (our login name on the client). When we type the RETURN key, this default is sent.

Asking for a directory listing of a singe file causes a data connection to be established and used. This example follows the procedure we showed in Figures 27.4 and 27.5. The client asks its TCP for an ephemeral port number for its end of the data connection, and sends this port number (1174) to the server in a PORT command. We can also see that a single interactive user command (`dir`) becomes two FTP commands (PORT and LIST).

Figure 27.6 is the time line of the packet exchange across the control connection. (We have removed the establishment and termination of the control connection, along with all the window size advertisements.) We note in this figure where the data connection is opened, used, and then closed.

Figure 27.7 is the time line for the data connection. The times in this figure are from the same starting point as Figure 27.6. We have removed all window advertisements, but have left in the type-of-service field, to show that the data connection uses a different type-of-service (maximize throughput) than the control connection (minimize delay). (The TOS values are in Figure 3.2.)

In this time line the FTP server does the active open of the data connection, from port 20 (called `ftp-data`), to the port number from the PORT command (1174). Also in this example, where the server writes to the data connection, the server does the active close of the data connection, which tells the client when the listing is complete.

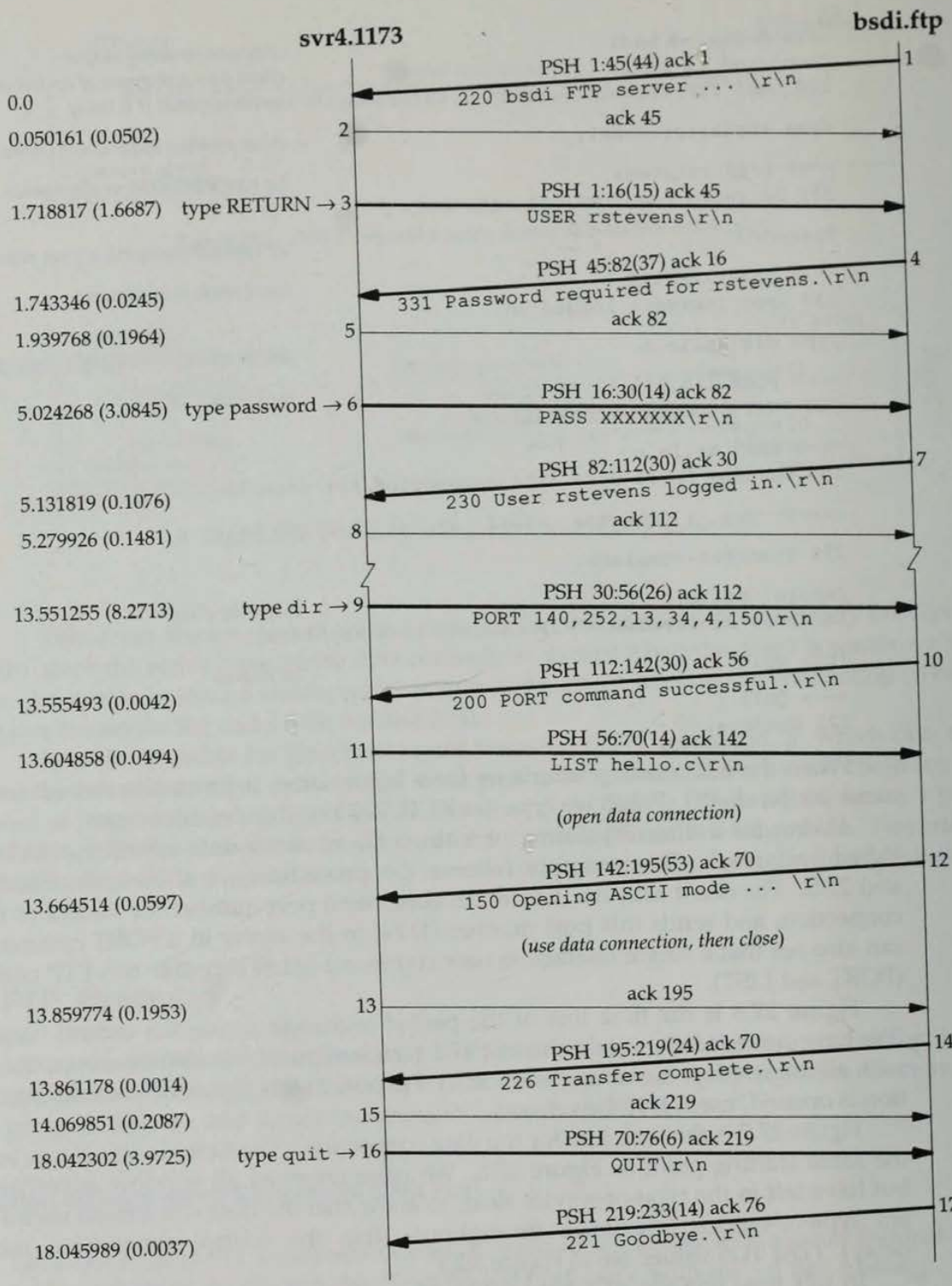


Figure 27.6 Control connection for FTP example.

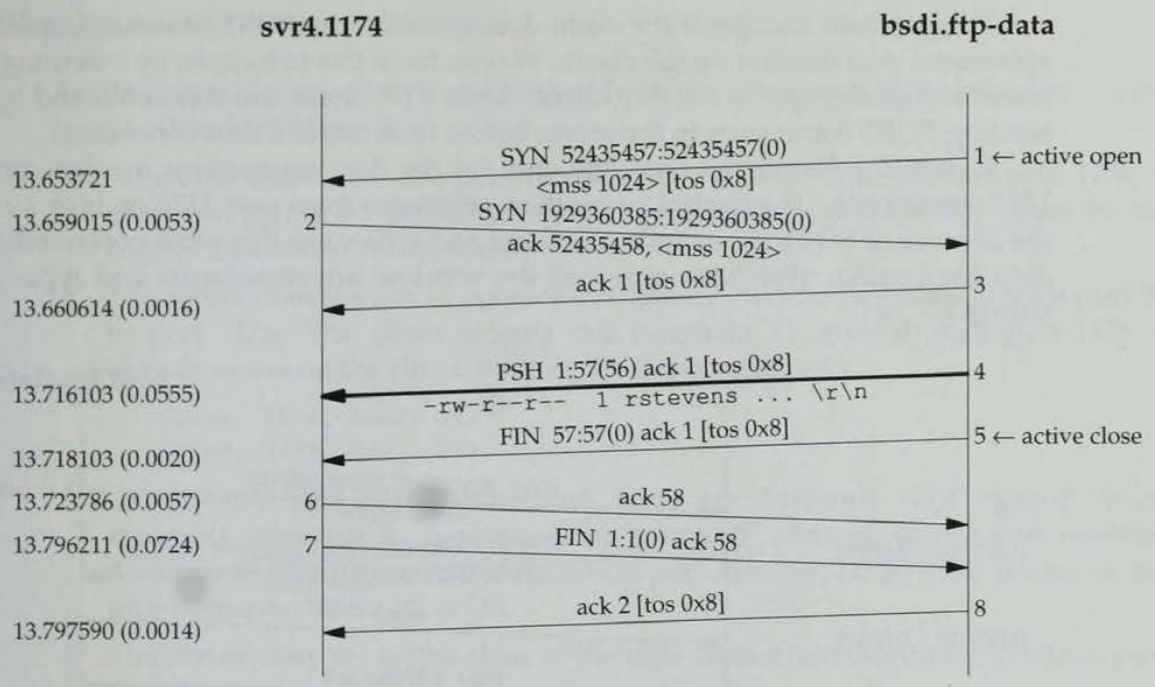


Figure 27.7 Data connection for FTP example.

Connection Management: Default Data Port

If the client does not send a PORT command to the server, to specify the port number for the client's end of the data connection, the server uses the same port number for the data connection that is being used for the control connection. This can cause problems for clients that use the stream mode (which the Unix FTP clients and server always use), as we show below.

> The Host Requirements RFC recommends that an FTP client using the stream mode send a PORT command to use a nondefault port number before each use of the data connection.

Returning to the previous example (Figure 27.6), what if we asked for another directory listing a few seconds after the first? The client would ask its kernel to choose another ephemeral port number (perhaps 1175) and the next data connection would be between `svr4` port 1175 and `bsdi` port 20. But in Figure 27.7 the server did the active close of the data connection, and we showed in Section 18.6 that the server won't be able to assign port 20 to the new data connection, because that local port number was used by an earlier connection that is still in the 2MSL wait state.

The server gets around this by specifying the `SO_REUSEADDR` option that we mentioned in Section 18.6. This lets it assign port 20 to the new connection, and the new connection will have a different foreign port number (1175) from the one that's in the 2MSL wait (1174), so everything is OK.

This scenario changes if the client does not send the PORT command, specifying an ephemeral port number on the client. We can force this to happen by executing the user command `sendport` to the FTP client. Unix FTP clients use this command to turn off sending PORT commands to the server before each use of a data connection.

Figure 27.8 shows the time line only for the data connections for two consecutive LIST commands. The control connection originates from port 1176 on host `svr4`, so in the absence of PORT commands, the client and server use this same port number for the data connection. (We have removed the window advertisements and type-of-service values.)

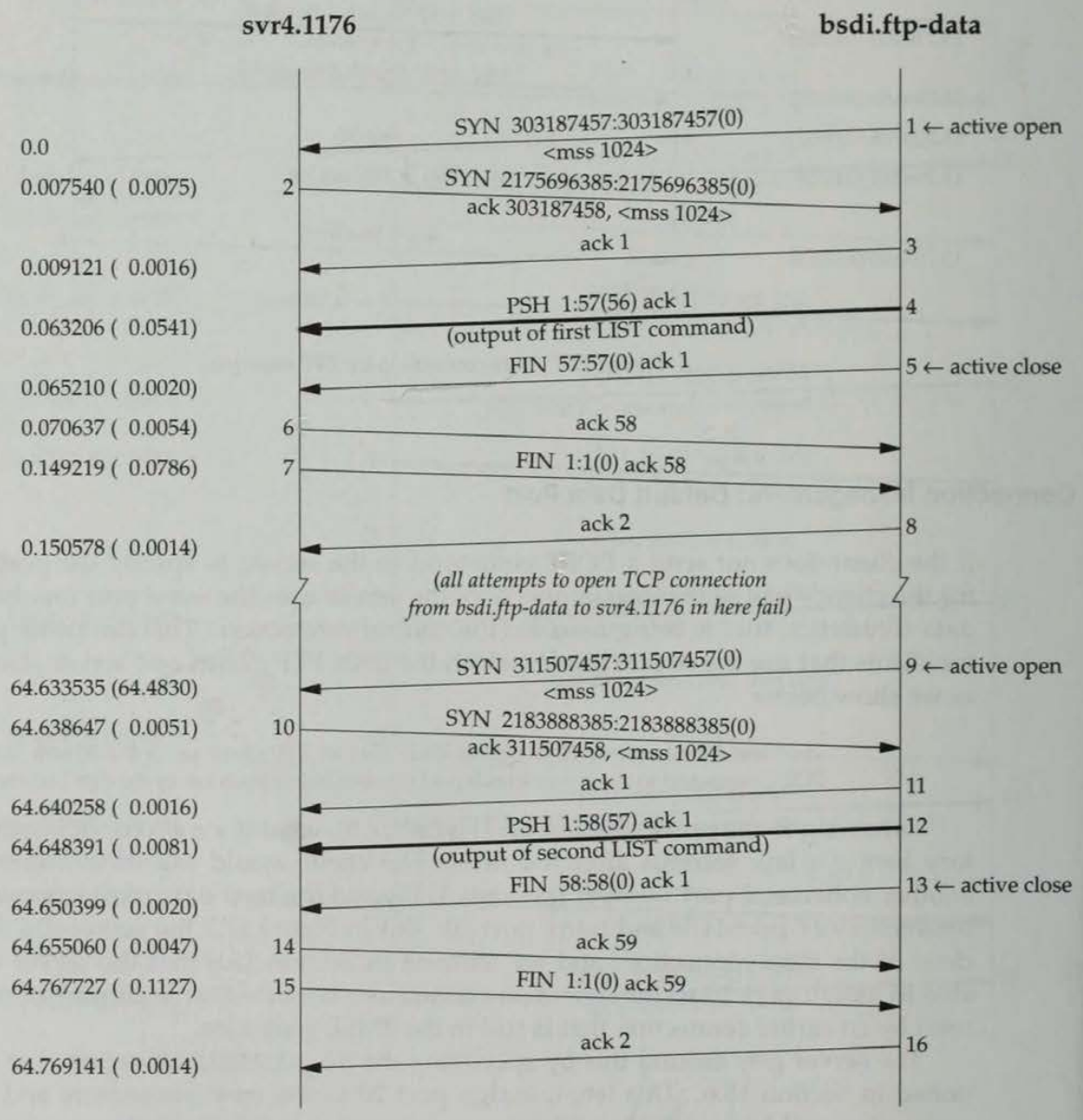


Figure 27.8 Data connection for two consecutive LIST commands.

The sequence of events is as follows.

1. The control connection is established from the client port 1176 to the server port 21. (We don't show this.)
2. When the client does the passive open for the data connection on port 1176, it must specify the `SO_REUSEADDR` option since that port is already in use by the control connection on the client.
3. The server does the active open of the data connection (segment 1) from port 20 to port 1176. The client accepts this (segment 2), even though port 1176 is already in use on the client, because the two socket pairs

   ```
   <svr4, 1176, bsdi, 21>
   <svr4, 1176, bsdi, 20>
   ```

   are different (the port numbers on `bsdi` are different). TCP demultiplexes incoming segments by looking at the source IP address, source port number, destination IP address, and destination port number, so as long as one of the four elements differs, all is OK.
4. The server does the active close of the data connection (segment 5), which puts the socket pair

   ```
   <svr4, 1176, bsdi, 20>
   ```

   in a 2MSL wait on the server.
5. The client sends another LIST command across the control connection. (We don't show this.) Before doing this the client does a passive open on port 1176 for its end of the data connection. The client must specify the `SO_REUSEADDR` option again, since the port number 1176 is already in use.
6. The server issues an active open for the data connection from port 20 to port 1176. Before doing this the server must specify `SO_REUSEADDR`, since the local port (20) is associated with a connection that is in the 2MSL wait, but from what we showed in Section 18.6, the connection won't succeed. The reason is that the socket pair for the connection request equals the socket pair from step 4 that is still in a 2MSL wait. The rules of TCP forbid the server from sending the SYN. There is no way for the server to override this 2MSL wait of the socket pair before reusing the same socket pair.

   At this point the BSD server retries the connection request every 5 seconds, up to 18 times, for a total of 90 seconds. We see that segment 9 succeeds about 1 minute later. (We mentioned in Chapter 18 that SVR4 uses an MSL of 30 seconds, for a 2MSL wait of 1 minute.) We don't see any SYNs from these failures in this time line because the active opens fail and the server's TCP doesn't even send a SYN.

The reason the Host Requirements RFC recommends using the PORT command is to avoid this 2MSL wait between successive uses of a data connection. By continually changing the port number on one end, the problem we just showed disappears.

Text File Transfer: NVT ASCII Representation or Image?

Let's verify that the transmission of a text file uses NVT ASCII by default. This time we don't specify the -d flag, so we don't see the client commands, but notice that the client still prints the server's responses:

```
sun % ftp bsdi
Connected to bsdi.
220 bsdi FTP server (Version 5.60) ready.
Name (bsdi:rstevens):                                   we type RETURN
331 Password required for rstevens.
Password:                                               we type our password
230 User rstevens logged in.
ftp> get hello.c                                        fetch a file
200 PORT command successful.
150 Opening ASCII mode data connection for hello.c (38 bytes).
226 Transfer complete.                                  server says file contains 38 bytes
local: hello.c remote: hello.c                          output by client
42 bytes received in 0.0037 seconds (11 Kbytes/s)       42 bytes across data connection
ftp> quit
221 Goodbye.
sun % ls -l hello.c
-rw-rw-r--  1 rstevens       38 Jul 18 08:48 hello.c    but file contains 38 bytes
sun % wc -l hello.c                                     count the lines in the file
       4 hello.c
```

Forty-two bytes are transferred across the data connection because the file contains four lines. Each Unix newline character (\n) is converted into the NVT ASCII 2-byte end-of-line sequence (\r\n) by the server for transmission, and then converted back by the client for storage.

Newer clients attempt to determine if the server is of the same system type, and if so, transfer files in binary (image file type) instead of ASCII. This helps in two ways.

1. The sender and receiver don't have to look at every byte (a big savings).
2. Fewer bytes are transferred if the host operating system uses fewer bytes for the end-of-line than the 2-byte NVT ASCII sequence (a smaller savings).

We can see this optimization using a BSD/386 client and server. We'll enable the debug mode, to see the client FTP commands:

```
bsdi % ftp -d slip                                      specify -d to see client commands
Connected to slip.
220 slip FTP server (Version 5.60) ready.
Name (slip:rstevens):                                   we type RETURN
---> USER rstevens
331 Password required for rstevens.
Password:                                               we type our password
```

```
---> PASS XXXX
230 User rstevens logged in.
---> SYST                                   this is sent automatically by client
215 UNIX Type: L8 Version: BSD-199103       server's reply
Remote system type is UNIX.                 information output by client
Using binary mode to transfer files.        information output by client
ftp> get hello.c                            fetch a file
---> TYPE I                                 sent automatically by client
200 Type set to I.
---> PORT 140,252,13,66,4,84                port number = 4 × 256 + 84 = 1108
200 PORT command successful.
---> RETR hello.c
150 Opening BINARY mode data connection for hello.c (38 bytes).
226 Transfer complete.
38 bytes received in 0.035 seconds (1.1 Kbytes/s)   only 38 bytes this time
ftp> quit
---> QUIT
221 Goodbye.
```

After we login to the server, the client FTP automatically sends the SYST command, which the server responds to with its system type. If the reply begins with the string "`215 UNIX Type: L8`", and if the client is running on a Unix system with 8 bits per byte, binary mode (image) is used for all file transfers, unless changed by the user.

When we fetch the file `hello.c` the client automatically sends the command TYPE I to set the file type to image. Only 38 bytes are transferred across the data connection.

> The Host Requirements RFC says an FTP server must support the SYST command (it was an option in RFC 959). But the only systems used in the text (see inside front cover) that support it are BSD/386 and AIX 3.2.2. SunOS 4.1.3 and Solaris 2.x reply with 500 (command not understood). SVR4 has the extremely unsocial behavior of replying with 500 *and* closing the control connection!

Aborting A File Transfer: Telnet Synch Signal

We now look at how the FTP client aborts a file transfer from the server. Aborting a file transfer from the client to the server is easy—the client stops sending data across the data connection and sends an ABOR to the server on the control connection. Aborting a receive, however, is more complicated, because the client wants to tell the server to stop sending data immediately. We mentioned earlier that the Telnet synch signal is used, as we'll see in this example.

We'll initiate a receive and type our interrupt key after it has started. Here is the interactive session, with the initial login deleted:

```
ftp> get a.out                              fetch a large file
---> TYPE I                                 client and server are both 8-bit byte Unix systems
200 Type set to I.
---> PORT 140,252,13,66,4,99
200 PORT command successful.
```

```
---> RETR a.out
150 Opening BINARY mode data connection for a.out (28672 bytes).
^?                                          type our interrupt key
receive aborted                             output by client
waiting for remote to finish abort          output by client

426 Transfer aborted. Data connection closed.
226 Abort successful

1536 bytes received in 1.7 seconds (0.89 Kbytes/s)
```

After we type our interrupt key, the client immediately tells us it initiated the abort and is waiting for the server to complete. The server sends two replies: 426 and 226. Both replies are sent by the Unix server when it receives the urgent data from the client with the ABOR command.

Figures 27.9 and 27.10 show the time line for this session. We have combined the control connection (solid lines) and the data connection (dashed lines) to show the relationship between the two.

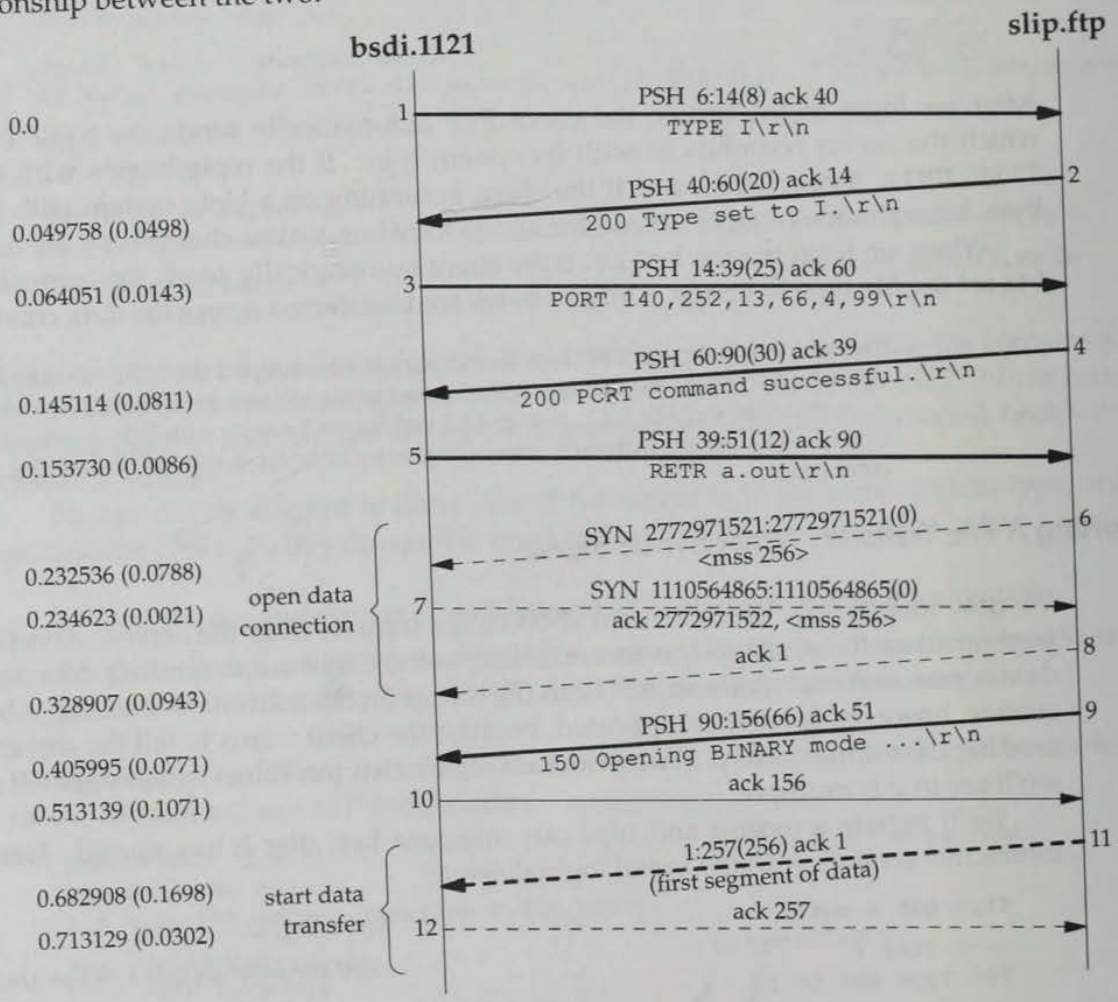


Figure 27.9 Aborting a file transfer (first half).

The first 12 segments in Figure 27.9 are what we expect. The commands and replies across the control connection set up the file transfer, the data connection is opened, and the first segment of data is sent from the server to the client.

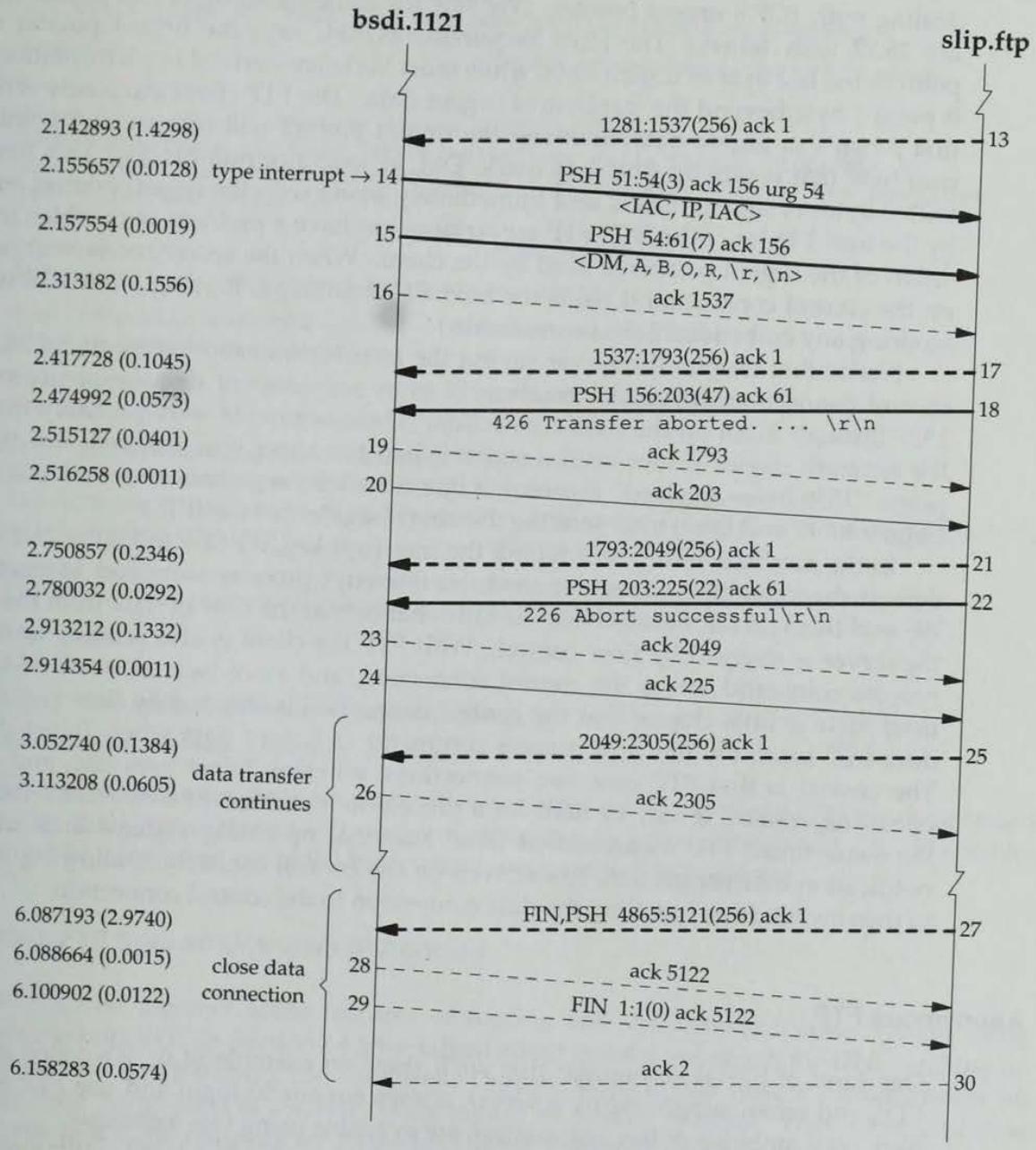


Figure 27.10 Aborting a file transfer (second half).

In Figure 27.10, segment 13 is the receipt of the sixth data segment from the server on the data connection, followed by segment 14, which is generated by our typing the interrupt key. Ten bytes are sent by the client to abort the transfer:

<IAC, IP, IAC, DM, A, B, 0, R, \r, \n>

We see two segments (14 and 15) because of the problem we detailed in Section 20.8 dealing with TCP's urgent pointer. (We saw the same handling of this problem in Figure 26.17 with Telnet.) The Host Requirements RFC says the urgent pointer should point to the last byte of urgent data, while most Berkeley-derived implementations have it point 1 byte beyond the last byte of urgent data. The FTP client purposely writes the first 3 bytes as urgent data, knowing the urgent pointer will (incorrectly) point to the next byte that is written (the data mark, DM, at sequence number 54). This first write with 3 bytes of urgent data is sent immediately, along with the urgent pointer, followed by the next 7 bytes. (The BSD FTP server does not have a problem with which interpretation of the urgent pointer is used by the client. When the server receives urgent data on the control connection it reads the next FTP command, looking for ABOR or STAT, ignoring any embedded Telnet commands.)

Notice that despite the server saying the transfer was aborted (segment 18, on the control connection), the client receives 14 more segments of data (sequence numbers 1537 through 5120) on the data connection. These segments were probably queued in the network device driver on the server when the abort was received, but the client prints "1536 bytes received" meaning it ignores all the segments of data that it receives (segments 17 and later) after sending the abort (segments 14 and 15).

In the case of a Telnet user typing the interrupt key, we saw in Figure 26.17 that by default the Unix client does not send the interrupt process command as urgent data. We said this was OK because there is little chance that the flow of data from the client to the server is stopped by flow control. With FTP the client is also sending an interrupt process command across the control connection, and since two connections are being used there is little chance that the control connection is stopped by flow control. Why does FTP send the interrupt process command as urgent data when Telnet does not? The answer is that FTP uses two connections, whereas Telnet uses one, and on some operating systems it may be hard for a process to monitor two connections for input at the same time. FTP assumes that these marginal operating systems at least provide notification that urgent data has arrived on the control connection, allowing the server to then switch from handling the data connection to the control connection.

Anonymous FTP

One form of FTP is so popular that we'll show an example of it. It's called *anonymous FTP*, and when supported by a server, allows anyone to login and use FTP to transfer files. Vast amounts of free information are available using this technique.

How to find which site has what you're looking for is a totally different problem. We mention it briefly in Section 30.4.

We'll use anonymous FTP to the site `ftp.uu.net` (a popular anonymous FTP site) to fetch the errata file for this book. To use anonymous FTP we login with the username of "anonymous" (you learn to spell it correctly after a few times). When prompted for a password we type our electronic mail address.

```
sun % ftp ftp.uu.net
Connected to ftp.uu.net.
220 ftp.UU.NET FTP server (Version 2.0WU(13) Fri Apr 9 20:44:32 EDT 1993) ready.
Name (ftp.uu.net:rstevens): anonymous
331 Guest login ok, send your complete e-mail address as password.
Password:                              we type rstevens@noao.edu; it's not echoed
230-
230-                Welcome to the UUNET archive.
230-    A service of UUNET Technologies Inc, Falls Church, Virginia
230-    For information about UUNET, call +1 703 204 8000, or see the files
230-    in /uunet-info
                                       more greeting lines
230 Guest login ok, access restrictions apply.
ftp> cd published/books                change to the desired directory
250 CWD command successful.
ftp> binary                            we'll transfer a binary file
200 Type set to I.
ftp> get stevens.tcpipiv1.errata.Z     fetch the file
200 PORT command successful.
150 Opening BINARY mode data connection for stevens.tcpipiv1.errata.Z (105 bytes).
226 Transfer complete.                 (you may get a different file size)
local: stevens.tcpipiv1.errata.Z remote: stevens.tcpipiv1.errata.Z
105 bytes received in 4.1 seconds (0.83 Kbytes/s)
ftp> quit
221 Goodbye.
sun % uncompress stevens.tcpipiv1.errata.Z
sun % more stevens.tcpipiv1.errata
```

The `uncompress` is because many files available for anonymous FTP are compressed using the Unix `compress`(1) program, resulting in a file extension of `.Z`. These files must be transferred using the binary file type, not the ASCII file type.

Anonymous FTP from an Unknown IP Address

We can tie together some features of routing and the Domain Name System using anonymous FTP. In Section 14.5 we talked about pointer queries in the DNS—taking an IP address and returning the hostname. Unfortunately not all system administrators set up their name servers correctly with regard to pointer queries. They often add new hosts to the file required for name-to-address mapping, but forget to add them to the file for address-to-name mapping. We often see this with `traceroute`, when it prints an IP address instead of a hostname.

Some anonymous FTP servers require that the client have a valid domain name. This allows the server to log the domain name of the host that's doing the transfer. Since the only client identification the server receives in the IP datagram from the client

is the IP address of the client, the server can call the DNS to do a pointer query, and obtain the domain name of the client. If the name server responsible for the client host is not set up correctly, this pointer query can fail.

To see this error we'll do the following steps.

1. Change the IP address of our host `slip` (see the figure on the inside front cover) to 140.252.13.67. This is a valid IP address for the author's subnet, but not entered into the name server for the `noao.edu` domain.
2. Change the destination IP address of the SLIP link on `bsdi` to 140.252.13.67.
3. Add a routing table entry on `sun` that directs datagrams for 140.252.13.67 to the router `bsdi`. (Recall our discussion of this routing table in Section 9.2.)

Our host `slip` is still reachable across the Internet, because we saw in Section 10.4 that the routers `gateway` and `netb` just sent any datagram destined for the subnet 140.252.13 to the router `sun`. Our router `sun` knows what to do with these datagrams from the routing table entry we made in step 3 above. What we have created is a host with complete Internet connectivity, but without a valid domain name. That is, a pointer query for the IP address 140.252.13.67 will fail.

We now use anonymous FTP to a server that we know requires a valid domain name:

```
slip % ftp ftp.uu.net
Connected to ftp.uu.net.

220 ftp.UU.NET FTP server (Version 2.0WU(13) Fri Apr 9 20:44:32 EDT 1993) ready.
Name (ftp.uu.net:rstevens): anonymous

530- Sorry, we're unable to map your IP address 140.252.13.67 to a hostname
530- in the DNS.  This is probably because your nameserver does not have a
530- PTR record for your address in its tables, or because your reverse
530- nameservers are not registered.  We refuse service to hosts whose
530- names we cannot resolve.  If this is simply because your nameserver is
530- hard to reach or slow to respond then try again in a minute or so, and
530- perhaps our nameserver will have your hostname in its cache by then.
530- If not, try reaching us from a host that is in the DNS or have your
530- system administrator fix your servers.
530 User anonymous access denied..

Login failed.
Remote system type is UNIX.
Using binary mode to transfer files.

ftp> quit
221 Goodbye.
```

The error reply from the server is self-explanatory.

27.4 Summary

FTP is the Internet standard for file transfer. Unlike most other TCP applications, it uses two TCP connections between the client and server—a control connection that is left up for the duration of the client–server session, and a data connection that is created and deleted as necessary.

The connection management used by FTP for the data connection has let us examine in more detail the connection management requirements of TCP. We saw the interaction of TCP's 2MSL wait state on clients that don't issue PORT commands.

FTP uses NVT ASCII from Telnet for all commands and replies across the control connection. The default data transfer mode is often NVT ASCII also. We saw that newer Unix clients automatically send a command to see if the server is an 8-bit byte Unix host, and if so, use binary mode for all file transfers, which is more efficient.

We also showed an example of anonymous FTP, a popular form of software distribution on the Internet.

Exercises

27.1 In Figure 27.8, what would change if the client did the active open of the second data connection instead of the server?

27.2 In the FTP client examples in this chapter we added the notation to lines such as

```
local: hello.c remote: hello.c
42 bytes received in 0.0037 seconds (11 Kbytes/s)
```

that the lines were output by the client. Without looking at the source code, how are we certain these are not from the server?

28

SMTP: Simple Mail Transfer Protocol

28.1 Introduction

Electronic mail (e-mail) is undoubtedly one of the most popular applications. [Caceres et al. 1991] shows that about one-half of all TCP connections are for the *Simple Mail Transfer Protocol*, SMTP. (On a byte count basis, FTP connections carry more data.) [Paxson 1993] found that the average mail message contains around 1500 bytes of data, but some messages contain megabytes of data, because electronic mail is sometimes used to send files.

Figure 28.1 shows an outline of e-mail exchange using TCP/IP.

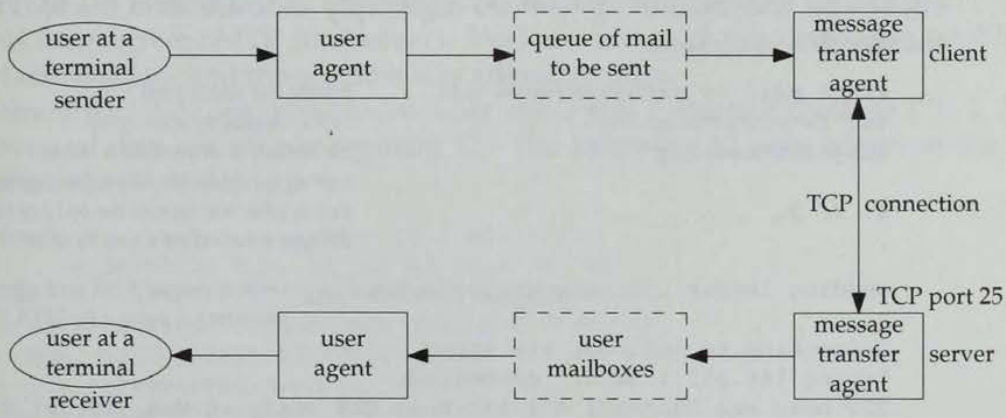


Figure 28.1 Outline of Internet electronic mail.

Users deal with a *user agent*, of which there are a multitude to choose from. Popular user agents for Unix include MH, Berkeley Mail, Elm, and Mush.

The exchange of mail using TCP is performed by a *message transfer agent* (MTA). The most common MTA for Unix systems is Sendmail. Users normally don't deal with the MTA. It is the responsibility of the system administrator to set up the local MTA. Users often have a choice, however, for their user agent.

This chapter examines the exchange of electronic mail between the two MTAs using TCP. We do not look at the operation or design of user agents.

RFC 821 [Postel 1982] specifies the SMTP protocol. This is how two MTAs communicate with each other across a single TCP connection. RFC 822 [Crocker 1982] specifies the format of the electronic mail message that is transmitted using RFC 821 between the two MTAs.

28.2 SMTP Protocol

The communication between the two MTAs uses NVT ASCII. Commands are sent by the client to the server, and the server responds with numeric reply codes and optional human-readable strings. This is similar to what we saw with FTP in the previous chapter.

There are a small number of commands that the client can send to the server: less than a dozen. (By comparison, FTP has more than 40 commands.) Rather than describing each one, we'll start with a simple example to show what happens when we send mail.

Simple Example

We'll send a simple one-line message and watch the SMTP connection. We invoke our user agent with the `-v` flag, which is passed to the mail transport agent (Sendmail in this case). This MTA displays what is sent and received across the SMTP connection when this flag is specified. Lines beginning with >>> are commands sent by the SMTP client, and lines beginning with a 3-digit reply code are from the SMTP server. Here is the interactive session:

```
sun % mail -v rstevens@noao.edu          invoke our user agent
To: rstevens@noao.edu                    this is output by user agent
Subject: testing                         we're then prompted for a subject
                                         user agent adds one blank line between headers and body
1, 2, 3.                                 this is what we type as the body of the message
.                                        we type a period on a line by itself to say we're done

Sending letter ... rstevens@noao.edu...  verbose output from user agent
                                         following is output by MTA (Sendmail)
Connecting to mailhost via ether...
Trying 140.252.1.54...  connected.
220 noao.edu Sendmail 4.1/SAG-Noao.G89 ready at Mon, 19 Jul 93 12:47:34 MST

>>> HELO sun.tuc.noao.edu.
250 noao.edu Hello sun.tuc.noao.edu., pleased to meet you

>>> MAIL From:<rstevens@sun.tuc.noao.edu>
250 <rstevens@sun.tuc.noao.edu>... Sender ok
```

```
>>> RCPT To:<rstevens@noao.edu>
250 <rstevens@noao.edu>... Recipient ok

>>> DATA
354 Enter mail, end with "." on a line by itself

>>> .
250 Mail accepted

>>> QUIT
221 noao.edu delivering mail

rstevens@noao.edu... Sent
sent.                                    this is output by user agent
```

Only five SMTP commands are used to send the mail: HELO, MAIL, RCPT, DATA, and QUIT.

We type `mail` to invoke our user agent. We're then prompted for a subject, and after typing that, we type the body of the message. Typing a period on a line by itself completes the message and the user agent passes the mail to the MTA for delivery.

The client does the active open to TCP port 25. When this returns, the client waits for a greeting message (reply code 220) from the server. This server's response must start with the fully qualified domain name of the server's host: `noao.edu` in this example. (Normally the text that follows the numeric reply code is optional. Here the domain name is required. The text beginning with `Sendmail` is optional.)

Next the client identifies itself with the HELO command. The argument must be the fully qualified domain name of the client host: `sun.tuc.noao.edu`.

The MAIL command identifies the originator of the message. The next command, RCPT, identifies the recipient. More than one RCPT command can be issued if there are multiple recipients.

The contents of the mail message are sent by the client using the DATA command. The end of the message is specified by the client sending a line containing just a period. The final command, QUIT, terminates the mail exchange.

Figure 28.2 is a time line of the SMTP connection between the sender SMTP (the client) and the receiver SMTP (the server). We have removed the connection establishment and termination, and the window size advertisements.

The amount of data we typed to our user agent was a one-line message ("1, 2, 3."), yet 393 bytes of data are sent in segment 12. The following 12 lines comprise the 393 bytes that are sent by the client:

```
Received: by sun.tuc.noao.edu. (4.1/SMI-4.1)
        id AA00502; Mon, 19 Jul 93 12:47:32 MST
Message-Id: <9307191947.AA00502@sun.tuc.noao.edu.>
From: rstevens@sun.tuc.noao.edu (Richard Stevens)
Date: Mon, 19 Jul 1993 12:47:31 -0700
Reply-To: rstevens@noao.edu
X-Phone: +1 602 676 1676
X-Mailer: Mail User's Shell (7.2.5 10/14/92)
To: rstevens@noao.edu
Subject: testing

1, 2, 3.
```

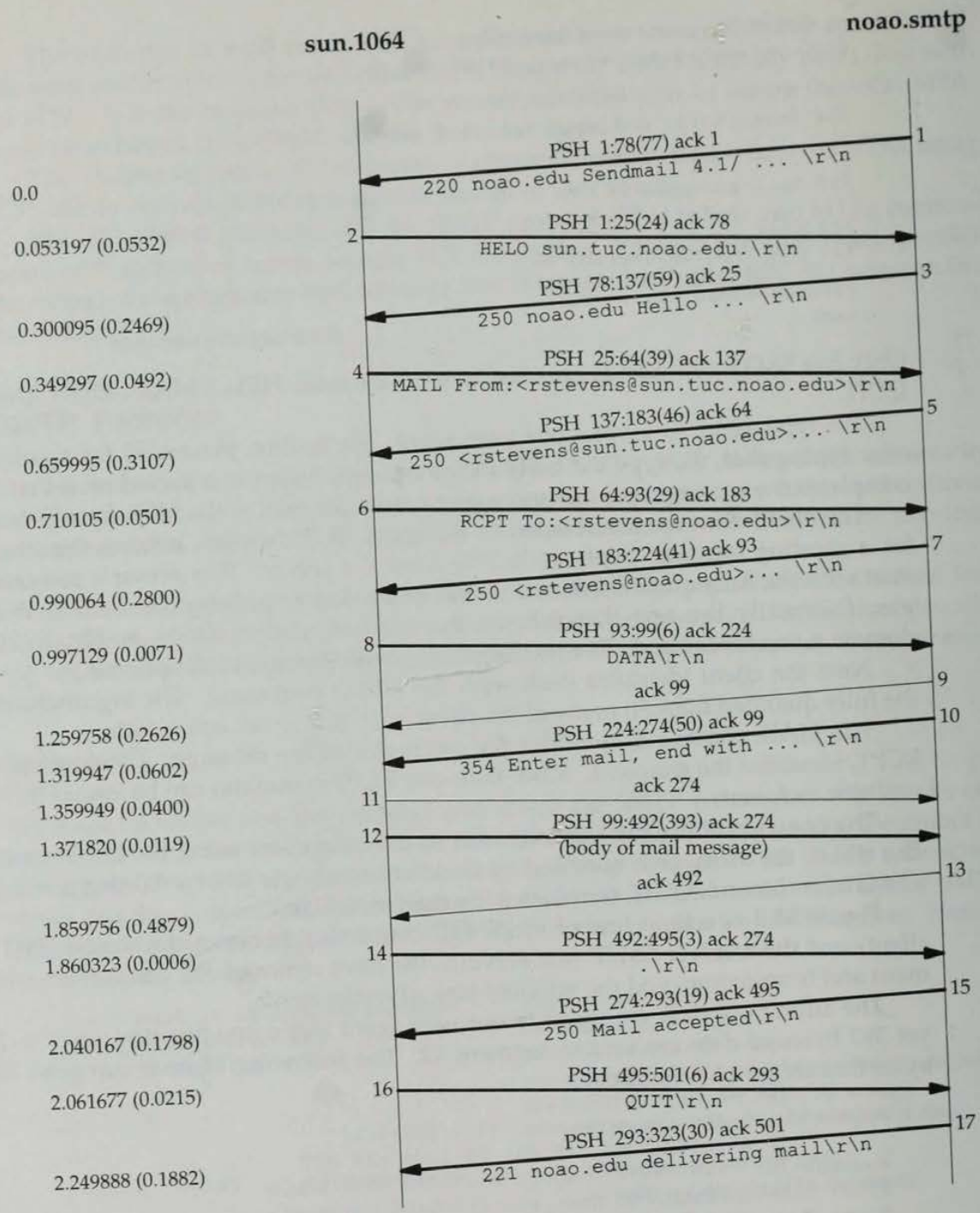


Figure 28.2 Basic SMTP mail delivery.

The first three lines, `Received:` and `Message-Id:`, are added by the MTA, and the next nine are generated by the user agent.

SMTP Commands

The minimal SMTP implementation supports eight commands. We saw five of them in the previous example: HELO, MAIL, RCPT, DATA, and QUIT.

The RSET command aborts the current mail transaction and causes both ends to reset. Any stored information about sender, recipients, or mail data is discarded.

The VRFY command lets the client ask the sender to verify a recipient address, without sending mail to the recipient. It's often used by a system administrator, by hand, for debugging mail delivery problems. We'll show an example of this in the next section.

The NOOP command does nothing besides force the server to respond with an OK reply code (200).

There are additional, optional commands. EXPN expands a mailing list, and is often used by the system administrator, similar to VRFY. Indeed, most versions of Sendmail handle the two identically.

> Version 8 of Sendmail in 4.4BSD no longer handles the two identically. VRFY does not expand aliases and doesn't follow `.forward` files.

The TURN command lets the client and server switch roles, to send mail in the reverse direction, without having to take down the TCP connection and create a new one. (Sendmail does not support this command.) There are three other commands (SEND, SOML, and SAML), which are rarely implemented, that replace the MAIL command. These three allow combinations of the mail being delivered directly to the user's terminal (if logged in), or sent to the recipient's mailbox.

Envelopes, Headers, and Body

Electronic mail is composed of three pieces.

1. The *envelope* is used by the MTAs for delivery. In our example the envelope was specified by the two SMTP commands:

   ```
   MAIL From:<rstevens@sun.tuc.noao.edu>
   RCPT To:<rstevens@noao.edu>
   ```

   RFC 821 specifies the contents and interpretation of the envelope, and the protocol used to exchange mail across a TCP connection.

2. *Headers* are used by the user agents. We saw nine header fields in our example: `Received`, `Message-Id`, `From`, `Date`, `Reply-To`, `X-Phone`, `X-Mailer`, `To`, and `Subject`. Each header field contains a name, followed by a colon, followed by the field value. RFC 822 specifies the format and interpretation of the header fields. (Headers beginning with an `X-` are user-defined fields. The others are defined by RFC 822.) Long header fields, such as `Received` in the example, are folded onto multiple lines, with the additional lines starting with white space.

3. The *body* is the content of the message from the sending user to the receiving user. RFC 822 specifies the body as lines of NVT ASCII text. When transferred

using the DATA command, the headers are sent first, followed by a blank line, followed by the body. Each line transferred using the DATA command must be less than 1000 bytes.

The user agent takes what we specify as the body, adds some headers, and passes the result to the MTA. The MTA adds a few headers, adds the envelope, and sends the result to another MTA.

The term *content* is often used to describe the combination of headers and the body. The content is sent by the client with the DATA command.

Relay Agents

The first line of informational output by our local MTA in our example is "Connecting to mailhost via ether." This is because the author's system has been configured to send all nonlocal outgoing mail to a relay machine for delivery.

This is done for two reasons. First, it simplifies the configuration of all MTAs other than the relay system's MTA. (Configuring an MTA is not simple, as anyone who has ever worked with Sendmail can attest to.) Second, it allows one system at an organization to act as the mail hub, possibly hiding all the individual systems.

In this example the relay system has a hostname of `mailhost` in the local domain (`.tuc.noao.edu`) and all the individual systems are configured to send their mail to this host. We can execute the `host` command to see how this name is defined to the DNS:

```
sun % host mailhost
mailhost.tuc.noao.edu   CNAME   noao.edu          canonical name
noao.edu                A       140.252.1.54      its real IP address
```

If the host used as the relay changes in the future, only its DNS name need change—the mail configuration of all the individual systems does not change.

Most organizations are using relay systems today. Figure 28.3 is a revised picture of Internet mail (Figure 28.2), taking into account that both the sending host and the final receiving host probably use a relay host.

In this scenario there are four MTAs between the sender and receiver. The local MTA on the sender's host just delivers the mail to its relay MTA. (This relay MTA could have a hostname of `mailhost` in the organization's domain.) This communication uses SMTP across the organization's local internet. The relay MTA in the sender's organization then sends the mail to the receiving organization's relay MTA across the Internet. This other relay MTA then delivers the mail to the receiver's host, by communication with the local MTA on the receiver's host. All the MTAs in this example use SMTP, although the possibility exists for other protocols to be used.

NVT ASCII

One feature of SMTP is that it uses NVT ASCII for everything: the envelope, the headers, and the body. As we said in Section 26.4, this is a 7-bit character code, transmitted as 8-bit bytes, with the high-order bit set to 0.

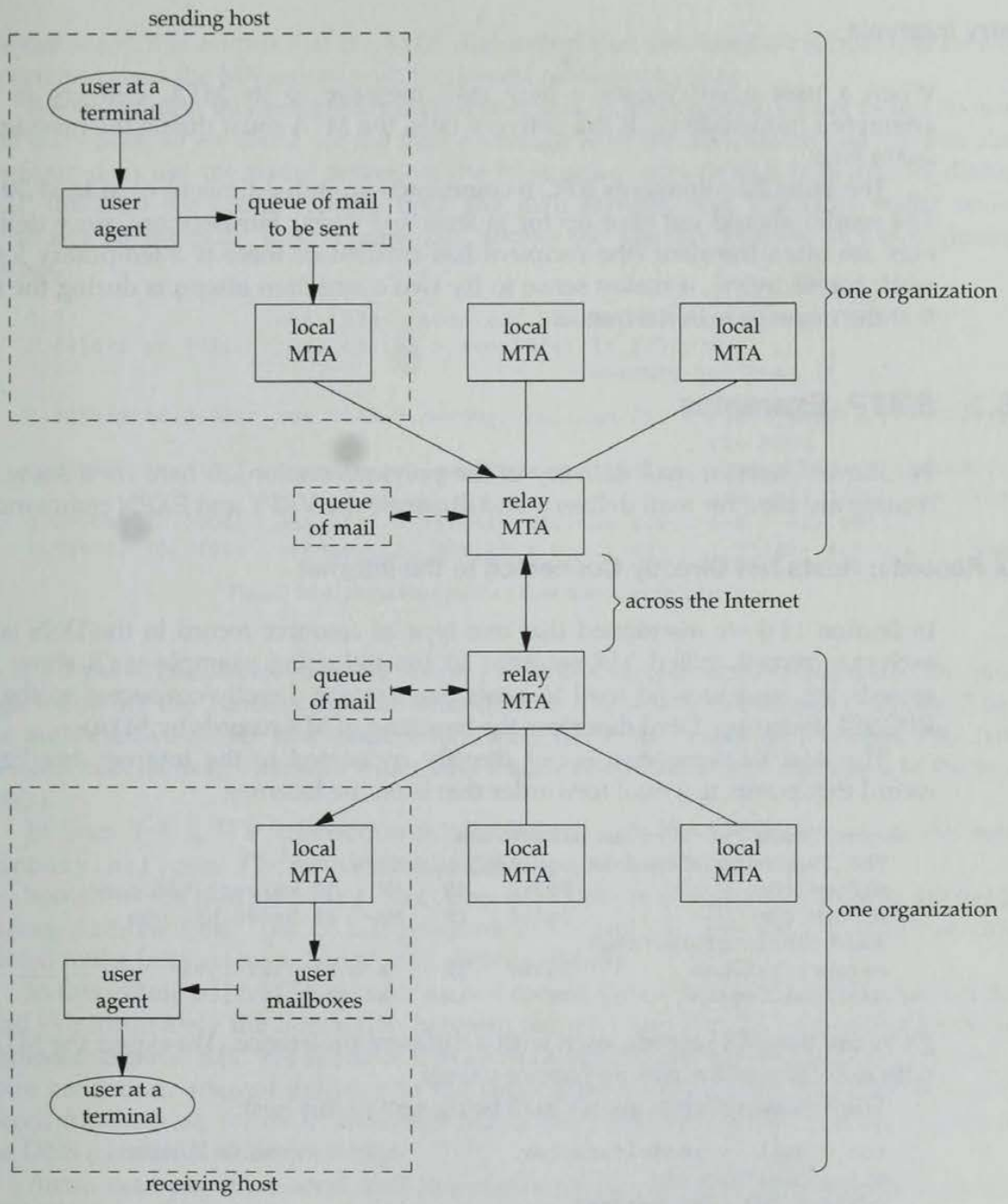


Figure 28.3 Internet electronic mail, with a relay system at both ends.

In Section 28.4 we discuss some newer features of Internet mail, extended SMTP and multimedia mail (MIME), that allow the sending and receiving of data such as audio and video. We'll see that MIME works with NVT ASCII for the envelope, headers, and body, with changes required only in the user agents.

Retry Intervals

When a user agent passes a new mail message to its MTA, delivery is normally attempted immediately. If the delivery fails, the MTA must queue the message and try again later.

The Host Requirements RFC recommends an initial timeout of at least 30 minutes. The sender should not give up for at least 4–5 days. Furthermore, since delivery failures are often transient (the recipient has crashed or there is a temporary loss of network connectivity), it makes sense to try two connection attempts during the first hour that the message is in the queue.

28.3 SMTP Examples

We showed normal mail delivery in the previous section, so here we'll show how MX records are used for mail delivery, and illustrate the VRFY and EXPN commands.

MX Records: Hosts Not Directly Connected to the Internet

In Section 14.6 we mentioned that one type of resource record in the DNS is the mail exchange record, called MX records. In the following example we'll show how MX records are used to send mail to hosts that are not directly connected to the Internet. RFC 974 [Partridge 1986] describes the handling of MX records by MTAs.

The host `mlfarm.com` is not directly connected to the Internet, but has an MX record that points to a mail forwarder that is on the Internet:

```
sun % host -a -v -t mx mlfarm.com
The following answer is not authoritative:
mlfarm.com              86388   IN   MX   10 mercury.hsi.com
mlfarm.com              86388   IN   MX   15 hsi86.hsi.com
Additional information:
mercury.hsi.com         86388   IN    A   143.122.1.91
hsi86.hsi.com          172762   IN    A   143.122.1.6
```

There are two MX records, each with a different preference. We expect the MTA to start with the lower of the two preference values.

The following script shows mail being sent to this host:

```
sun % mail -v ron@mlfarm.com          -v flag to see what the MTA does
To: ron@mlfarm.com
Subject: MX test message

                                      the body of the message is typed here (not shown)
.                                     period on a line by itself to terminate message
Sending letter ... ron@mlfarm.com...
Connecting to mlfarm.com via tcp...
mail exchanger is mercury.hsi.com     the MX records are found
Trying 143.122.1.91...  connected.   first tries the one with lower preference
220 mercury.hsi.com ...
                                      remainder is normal SMTP mail transfer
```

We can see in this output that the MTA discovered that the destination host had an MX record and used the MX record with the lowest preference value.

Before running this example from the host sun, it was configured not to use its normal relay host, so we could see the mail exchange with the destination host. It was also configured to use the name server on the host noao.edu (which is across its dialup SLIP link), so we could capture both the mail transfer and the DNS traffic using tcpdump on the SLIP link. Figure 28.4 shows the starting portion of the tcpdump output.

```
1  0.0                sun.1624 > noao.edu.53: 2+ MX? mlfarm.com. (28)
2  0.445572 (0.4456)  noao.edu.53 > sun.1624: 2* 2/0/2 MX
                                                mercury.hsi.com. 10 (113)

3  0.505739 (0.0602)  sun.1143 > mercury.hsi.com.25: S 1617536000:1617536000(0)
                                                      win 4096
4  0.985428 (0.4797)  mercury.hsi.com.25 > sun.1143: S 1832064000:1832064000(0)
                                                      ack 1617536001 win 16384
5  0.986003 (0.0006)  sun.1143 > mercury.hsi.com.25: . ack 1 win 4096
6  1.735360 (0.7494)  mercury.hsi.com.25 > sun.1143: P 1:90(89) ack 1 win 16384
```

Figure 28.4 Sending mail to a host that uses MX records.

In line 1 the MTA queries its name server for an MX record for mlfarm.com. The plus sign following the 2 means the recursion-desired flag is set. The response in line 2 has the authoritative bit set (the asterisk following the 2) and contains 2 answer RRs (the two MX host names), 0 authority RRs, and 2 additional RRs (the IP addresses of the two hosts).

In lines 3–5 a TCP connection is established with the SMTP server on the host mercury.hsi.com. The server's initial 220 response is shown in line 6.

Somehow the host mercury.hsi.com must deliver this mail message to the destination, mlfarm.com. The UUCP protocols are a popular way for a system not connected to the Internet to exchange mail with its MX site.

In this example the MTA asks for an MX record, gets a positive result, and sends the mail. Unfortunately the interaction between an MTA and the DNS can differ between implementations. RFC 974 specifies that an MTA should ask for MX records first, and if none are found, attempt delivery to the destination host (i.e., ask the DNS for an A record for the host, for its IP address). MTAs must also deal with CNAME records in the DNS (canonical names).

As an example, if we send mail to rstevens@mailhost.tuc.noao.edu from a BSD/386 host, the following steps are executed by the MTA (Sendmail).

1. Sendmail asks the DNS for CNAME records for mailhost.tuc.noao.edu. We see that a CNAME record exists:

```
sun % host -t cname mailhost.tuc.noao.edu
mailhost.tuc.noao.edu   CNAME   noao.edu
```

2. A DNS query is issued for CNAME records for noao.edu and the response says none exist.

3. Sendmail then asks the DNS for MX records for noao.edu and gets one MX record:

```
sun % host -t mx noao.edu
noao.edu                MX      noao.edu
```

4. Sendmail queries the DNS for an A record (IP address) for noao.edu and gets back the value 140.252.1.54. (This A record was probably returned by the name server for noao.edu as an additional RR with the MX reply in step 3.)
5. An SMTP connection is initiated to 140.252.1.54 and the mail is sent.

A CNAME query is not tried for the data returned in the MX record (noao.edu). The data in the MX record cannot be an alias—it must be the name of a host that has an A record.

> The version of Sendmail distributed with SunOS 4.1.3 that uses the DNS only queries for MX records, and gives up if an MX record isn't found.

MX Records: Hosts That Are Down

Another use of MX records is to provide an alternative mail receiver when the destination host is down. If we look at the DNS entry for our host sun we see that it has two MX records:

```
sun % host -a -v -t mx sun.tuc.noao.edu
sun.tuc.noao.edu        86400     IN     MX      0 sun.tuc.noao.edu
sun.tuc.noao.edu        86400     IN     MX     10 noao.edu
Additional information:
sun.tuc.noao.edu        86400     IN      A     140.252.1.29
sun.tuc.noao.edu        86400     IN      A     140.252.13.33
noao.edu                86400     IN      A     140.252.1.54
```

The MX record with the lowest preference indicates that direct delivery to the host itself should be tried first, and the next preference is to deliver the mail to the host noao.edu.

In the following script we send mail to ourself at the host sun.tuc.noao.edu, from the host vangogh.cs.berkeley.edu, after turning off the destination's SMTP server. When a connection request arrives for port 25, TCP should respond with an RST, since no process has a passive open pending for that port.

```
vangogh % mail -v rstevens@sun.tuc.noao.edu
A test to a host that's down.
.
EOT
rstevens@sun.tuc.noao.edu... Connecting to sun.tuc.noao.edu. (smtp)...
rstevens@sun.tuc.noao.edu... Connecting to noao.edu. (smtp)...
220 noao.edu ...
                                        remainder is normal SMTP mail transfer
```

We see that the MTA tries to contact sun.tuc.noao.edu and then gives up and contacts noao.edu instead.

Figure 28.5 is the `tcpdump` output that shows that TCP responds to the incoming SYNs with an RST.

```
1 0.0              vangogh.3873 > 140.252.1.29.25: S 2358303745:2358303745(0) ...
2 0.000621 (0.0006) 140.252.1.29.25 > vangogh.3873: R 0:0(0) ack 2358303746 win 0
3 0.300203 (0.2996) vangogh.3874 > 140.252.13.33.25: S 2358367745:2358367745(0) ...
4 0.300620 (0.0004) 140.252.13.33.25 > vangogh.3874: R 0:0(0) ack 2358367746 win 0
```

Figure 28.5 Attempt to connect to an SMTP server that is not running.

In line 1 `vangogh` sends a SYN to port 25 at the primary IP address for `sun`: 140.252.1.29. This is rejected in line 2. The SMTP client on `vangogh` then tries the next IP address for `sun`: 140.252.13.33 (line 3), and it also causes an RST to be returned (line 4).

The SMTP client doesn't try to differentiate between the different error returns from its active open on line 1, which is why it tries the other IP address on line 2. If the error had been something like "host unreachable" for the first attempt, it's possible that the second attempt could work.

If the reason the SMTP client's active open fails is because the server host is down, we would see the client retransmit the SYN to IP address 140.252.1.29 for a total of 75 seconds (similar to Figure 18.6), followed by the client sending another three SYNs to IP address 140.252.13.33 for another 75 seconds. After 150 seconds the client would move on to the next MX record with the higher preference.

VRFY and EXPN Commands

The VRFY command verifies that a recipient address is OK, without actually sending mail. EXPN is intended to expand a mailing list, without sending mail to the list. Many SMTP implementations (such as Sendmail) consider the two the same, but we mentioned that newer versions of Sendmail do differentiate between the two.

As a simple test we can connect to a newer version of Sendmail and see the difference. (We have removed the extraneous Telnet client output.)

```
sun % telnet vangogh.cs.berkeley.edu 25
220-vangogh.CS.Berkeley.EDU Sendmail 8.1C/6.32 ready at Tue, 3 Aug 1993 14:
59:12 -0700
220 ESMTP spoken here
helo bsdi.tuc.noao.edu
250 vangogh.CS.Berkeley.EDU Hello sun.tuc.noao.edu [140.252.1.29], pleased
to meet you
vrfy nosuchname
550 nosuchname... User unknown
vrfy rstevens
250 Richard Stevens <rstevens@vangogh.CS.Berkeley.EDU>
expn rstevens
250 Richard Stevens <rstevens@noao.edu>
```

First notice that we purposely typed the wrong hostname on the HELO command: `bsdi` instead of `sun`. Most SMTP servers take the IP address of the client and perform

a DNS pointer query (Section 14.5) and compare the hostnames. This allows the server to log the client connection based on the IP address, not the name that a user might have mistyped. Some servers respond with humorous messages, such as "You are a charlatan," or "why do you call yourself ...". We see in this example that this server just prints our real domain name from the pointer query along with our IP address.

We then type a VRFY command for an invalid name, and the server responds with a 550 error. Next we type a valid name, and the server responds with the username on the local host. Next we try the EXPN command and get a different response. The EXPN command determines that the mail for this user is being forwarded, and prints the forwarding address.

Many sites disable the VRFY and EXPN commands, sometimes for privacy, and sometimes in the belief that it's a security hole. For example, we can try these commands with the SMTP server at the White House:

```
sun % telnet whitehouse.gov 25
220 whitehouse.gov SMTP/smap Ready.

helo sun.tuc.noao.edu
250 (sun.tuc.noao.edu) pleased to meet you.

vrfy clinton
500 Command unrecognized

expn clinton
500 Command unrecognized
```

28.4 SMTP Futures

Changes are taking place with Internet mail. Recall the three pieces that comprise Internet mail: the envelope, headers, and body. New SMTP commands are being added that affect the envelope, non-ASCII characters can be used in the headers, and structure is being added to the body (MIME). In this section we consider the extensions to each of these three pieces in order.

Envelope Changes: Extended SMTP

RFC 1425 [Klensin et al. 1993a] defines the framework for adding extensions to SMTP. The result is called *extended SMTP* (ESMTP). As with other new features that we've described in the text, these changes are being added in a backward compatible manner, so that existing implementations aren't affected.

A client that wishes to use the new features initiates the session with the server by issuing a EHLO command, instead of HELO. A compatible server responds with a 250 reply code. This reply is normally multiline, with each line containing a keyword and an optional argument. These keywords specify the SMTP extensions supported by the server. New extensions will be described in an RFC and will be registered with the IANA. (In a multiline reply all lines except the last have a hyphen after the numeric reply code. The last line has a space after the numeric reply code.)

We'll show the initial connection to four SMTP servers, three of which support extended SMTP. We connect to them using Telnet, but have removed the extraneous Telnet client output.

```
sun % telnet vangogh.cs.berkeley.edu 25
220-vangogh.CS.Berkeley.EDU Sendmail 8.1C/6.32 ready at Mon, 2 Aug 1993 15:
47:48 -0700
220 ESMTP spoken here

ehlo sun.tuc.noao.edu
250-vangogh.CS.Berkeley.EDU Hello sun.tuc.noao.edu [140.252.1.29], pleased
to meet you
250-EXPN
250-SIZE
250 HELP
```

This server gives a multiline 220 reply for its greeting message. The extended commands listed in the 250 reply to the EHLO command are EXPN, SIZE, and HELP. The first and last are from the original RFC 821 specification, but they are optional commands. ESMTP servers state which of the optional RFC 821 commands they support, in addition to newer commands.

The SIZE keyword that this server supports is defined in RFC 1427 [Klensin, Freed, and Moore 1993]. It lets the client specify the size of the message in bytes on the MAIL FROM command line. This lets the server verify that it will accept a message of that size, before the client starts to send it. This command was added since the size of Internet mail messages is growing, with the support for message content other than ASCII lines (i.e., images, audio, etc.).

The next host also supports ESMTP. Notice that the 250 reply specifying that the SIZE keyword is supported contains an optional argument. This indicates that this server will accept a message size up to 461 Mbytes.

```
sun % telnet ymir.claremont.edu 25
220 ymir.claremont.edu -- Server SMTP (PMDF V4.2-13 #4220)

ehlo sun.tuc.noao.edu
250-ymir.claremont.edu
250-8BITMIME
250-EXPN
250-HELP
250-XADR
250 SIZE 461544960
```

The keyword 8BITMIME is from RFC 1426 [Klensin et al. 1993b]. This allows the client to add the keyword BODY to the MAIL FROM command, specifying whether the body contains NVT ASCII characters (the default) or 8-bit data. Unless the client receives the 8BITMIME keyword from the server in response to a EHLO command, the client is forbidden from sending any characters other than NVT ASCII. (When we talk about MIME in this section, we'll see that an 8-bit SMTP transport is *not* required by MIME.)

This server also advertises the XADR keyword. Any keyword that begins with an X refers to a local SMTP extension.

The next server also supports ESMTP, advertising the HELP and SIZE keywords that we've already seen. It also supports three local extensions that begin with an X.

```
sun % telnet dbc.mtview.ca.us 25
220 dbc.mtview.ca.us Sendmail 5.65/3.1.090690, it's Mon, 2 Aug 93 15:48:50
-0700

ehlo sun.tuc.noao.edu
250-Hello sun.tuc.noao.edu, pleased to meet you
250-HELP
250-SIZE
250-XONE
250-XVRB
250 XQUE
```

Finally we see what happens when the client tries to use ESMTP by issuing the EHLO command to a server that doesn't support it.

```
sun % telnet relay1.uu.net 25
220 relay1.UU.NET Sendmail 5.61/UUNET-internet-primary ready at Mon, 2 Aug
93 18:50:27 -0400

ehlo sun.tuc.noao.edu
500 Command unrecognized

rset
250 Reset state
```

Instead of receiving a 250 reply to the EHLO command, the client receives a 500 reply. The client should then issue the RSET command, followed by a HELO command.

Header Changes: Non-ASCII Characters

RFC 1522 [Moore 1993] specifies a way to send non-ASCII characters in RFC 822 message headers. The main use of this is to allow additional characters in the sender and receiver names, and in the subject.

The header fields can contain encoded words. They have the following format:

=? *charset* ? *encoding* ? *encoded-text* ?=

charset is the character set specification. Valid values are the two strings `us-ascii` and `iso-8859-X`, where `X` is a single digit, as in `iso-8859-1`.

encoding is a single character to specify the encoding method. Two values are supported.

1. `Q` encoding means *quoted-printable*, and is intended for Latin character sets. Most characters are sent as NVT ASCII (with the high-order bit set to 0, of course). Any character to be sent whose eighth bit is set is sent instead as three characters: first the character =, followed by two hexadecimal digits. For example, the character é (whose binary 8-bit value is `0xe9`) is sent as the three characters `=E9`. Spaces are always sent as either an underscore or the three characters `=20`. This encoding is intended for text that is mostly ASCII, with a few special characters.

2. B means base-64 encoding. Three consecutive bytes of text (24 bits) are encoded as four 6-bit values. The 64 NVT ASCII characters used to represent each of the possible 6-bit values are shown in Figure 28.6.

6-bit value	ASCII char	6-bit value	ASCII char	6-bit value	ASCII char	6-bit value	ASCII char
0	A	10	Q	20	g	30	w
1	B	11	R	21	h	31	x
2	C	12	S	22	i	32	y
3	D	13	T	23	j	33	z
4	E	14	U	24	k	34	0
5	F	15	V	25	l	35	1
6	G	16	W	26	m	36	2
7	H	17	X	27	n	37	3
8	I	18	Y	28	o	38	4
9	J	19	Z	29	p	39	5
a	K	1a	a	2a	q	3a	6
b	L	1b	b	2b	r	3b	7
c	M	1c	c	2c	s	3c	8
d	N	1d	d	2d	t	3d	9
e	O	1e	e	2e	u	3e	+
f	P	1f	f	2f	v	3f	/

Figure 28.6 Encoding of 6-bit values (base-64 encoding).

When the number of characters to encode is not a multiple of three, equal signs are used as the pad characters.

The following example of these two encodings is from RFC 1522:

```
From: =?US-ASCII?Q?Keith_Moore?= <moore@cs.utk.edu>
To: =?ISO-8859-1?Q?Keld_J=F8rn_Simonsen?= <keld@dkuug.dk>
CC: =?ISO-8859-1?Q?Andr=E9_?= Pirard <PIRARD@vm1.ulg.ac.be>
Subject: =?ISO-8859-1?B?SWYgeW91IGNhbiByZWFkIHRoaXMgeW8=?=
 =?ISO-8859-2?B?dSB1bmRlcnN0YW5kIHRoZSBleGFtcGxlLg==?=
```

A user agent capable of handling these headers would output:

```
From: Keith Moore <moore@cs.utk.edu>
To: Keld Jørn Simonsen <keld@dkuug.dk>
CC: André  Pirard <PIRARD@vm1.ulg.ac.be>
Subject: If you can read this you understand the example.
```

To see how base-64 encoding works, look at the first four encoded characters in the subject line: SWYg. Write out the 6-bit values for these four characters from Figure 28.6 (S=0x12, W=0x16, Y=0x18, and g=0x20) in binary:

```
010010 010110 011000 100000
```

Then regroup these 24 bits into three 8-bit bytes:

```
01001001 01100110 00100000
 =0x49    =0x66    =0x20
```

which are the ASCII representations for I, f, and a space.

Body Changes: Multipurpose Internet Mail Extensions (MIME)

We've said that RFC 822 specifies the body as lines of NVT ASCII text, with no structure. RFC 1521 [Borenstein and Freed 1993] defines extensions that allow structure in the body. This is called MIME, for *Multipurpose Internet Mail Extensions*.

MIME does *not* require any of the extensions that we've described previously in this section (extended SMTP or non-ASCII headers). MIME just adds some new headers (in accordance with RFC 822) that tell the recipient the structure of the body. The body can still be transmitted using NVT ASCII, regardless of the mail contents. While some of the extensions we've just described might be nice to have along with MIME—the extended SMTP SIZE command, since MIME messages can become large, and non-ASCII headers—these extensions are not required by MIME. All that's required to exchange MIME messages with another party is for both ends to have a user agent that understands MIME. No changes are required in any of the MTAs.

MIME defines the five new header fields:

```
Mime-Version:
Content-Type:
Content-Transfer-Encoding:
Content-ID:
Content-Description:
```

As an example, the following two header lines can appear in an Internet mail message:

```
Mime-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
```

The current MIME version is 1.0 and the content type is plain ASCII text, the default for Internet mail. The word `PLAIN` is considered a subtype of the content type (`TEXT`), and the string `charset=US-ASCII` is a parameter.

`Text` is just one of MIME's seven defined content types. Figure 28.7 summarizes the 16 different content types and subtypes defined in RFC 1521. Numerous parameters can be specified for certain content types and subtypes.

The content type and the transfer encoding used for the body are independent. The former is specified by the `Content-Type` header field, and the latter by the `Content-Transfer-Encoding` header field. There are five different encoding formats defined in RFC 1521.

1. `7bit`, which is NVT ASCII, the default.
2. `quoted-printable`, which we saw an example of earlier with non-ASCII headers. It is useful when only a small fraction of the characters have their eighth bit set.
3. `base64`, which we showed in Figure 28.6.
4. `8bit` containing lines of characters, some of which are non-ASCII and have their eighth bit set.
5. `binary` encoding, which is 8-bit data that need not contain lines.

Content-Type	Subtype	Description
text	plain	Unformatted text.
	richtext	Text with simple formatting, such as bold, italic, underline, and so on.
	enriched	A clarification, simplification, and refinement of richtext.
multipart	mixed	Multiple body parts to be processed sequentially.
	parallel	Multiple body parts that can be processed in parallel.
	digest	An electronic mail digest.
	alternative	Multiple body parts are present, all with identical semantic content.
message	rfc822	Content is another RFC 822 mail message.
	partial	Content is a fragment of a mail message.
	external-body	Content is a pointer to the actual message.
application	octet-stream	Arbitrary binary data.
	postscript	A PostScript program.
image	jpeg	ISO 10918 format.
	gif	CompuServe's Graphic Interchange Format.
audio	basic	Encoded using 8-bit ISDN μ-law format.
video	mpeg	ISO 11172 format.

Figure 28.7 MIME content types and subtypes.

Only the first three of these are valid for an RFC 821 MTA, since these three generate a body containing only NVT ASCII characters. Using extended SMTP with 8BITMIME support allows `8bit` encoding to be used.

Although the content type and encoding are independent, RFC 1521 recommends `quoted-printable` for `text` with non-ASCII data, and `base64` for `image`, `audio`, `video`, and `octet-stream` `application` data. This allows maximum interoperability with RFC 821 conformant MTAs. Also, the `multipart` and `message` content types must be encoded as `7bit`.

As an example of a `multipart` content type, Figure 28.8 shows a mail message from the RFC distribution list. The subtype is `mixed`, meaning each of the parts should be processed sequentially, and the boundary between the parts is the string `NextPart`, preceded by two hyphens at the start of a line.

Each boundary can be followed with a line specifying the header fields for the next part. Everything in the message before the first boundary is ignored, as is everything following the final boundary.

Since a blank line follows the first boundary, and not header fields, the content type of the data between the first and second boundaries is assumed to be `text/plain` with a character set of `us-ascii`. This is a textual description of the new RFC.

The second boundary, however, is followed by header fields. It specifies another `multipart` message, with a boundary of `OtherAccess`. The subtype is `alternative`, and two different alternatives are present. The first `OtherAccess` alternative is to fetch the RFC using electronic mail, and the second is to fetch it using anonymous FTP. A MIME user agent would list the two alternatives, allow us to choose one, and then automatically fetch a copy of the RFC using either mail or anonymous FTP.

```
To: rfc-dist@nic.ddn.mil
Subject: RFC1479 on IDPR Protocol
Mime-Version: 1.0
Content-Type: Multipart/Mixed; Boundary="NextPart"
Date: Fri, 23 Jul 93 12:17:43 PDT
From: "Joyce K. Reynolds" <jkrey@isi.edu>


--NextPart                                  the first boundary

A new Request for Comments is now available in online RFC libraries.

. . .                                       (details here on the new RFC)

Below is the data which will enable a MIME compliant Mail Reader
implementation to automatically retrieve the ASCII version
of the RFCs.

--NextPart                                  the second boundary
Content-Type: Multipart/Alternative; Boundary="OtherAccess"
                                            a nested multipart message with a new boundary

--OtherAccess
Content-Type:  Message/External-body;
        access-type="mail-server";
        server="mail-server@nisc.sri.com"

Content-Type: text/plain

SEND rfc1479.txt

--OtherAccess
Content-Type:   Message/External-body;
        name="rfc1479.txt";
        site="ds.internic.net";
        access-type="anon-ftp";
        directory="rfc"

Content-Type: text/plain


--OtherAccess--
--NextPart--                                the final boundary
```

Figure 28.8 Example of a MIME multipart message.

This section has been a brief overview of MIME. For additional details and examples of MIME, see RFC 1521 and [Rose 1993].

28.5 Summary

Electronic mail involves a user agent at both ends (the sender and receiver) and two or more message transfer agents. We can divide a mail message into three parts: the envelope, the headers, and the body. We've seen how all three parts are exchanged using SMTP, the Internet standard. All three are exchanged as NVT ASCII characters.

We've also looked at newer extensions for all three parts: extended SMTP for the envelope, non-ASCII headers, and the addition of structure to the body using MIME. The structure and encoding used by MIME allow arbitrary binary data to be exchanged, using existing 7-bit SMTP MTAs.

Exercises

28.1 Read RFC 822 to find out what a *domain literal* is. Try sending mail to yourself using one.

28.2 Excluding the connection establishment and termination, what is the minimum number of network round trips to send a small mail message?

28.3 TCP is a full-duplex protocol, yet SMTP uses TCP in a half-duplex fashion. The client sends a command then stops and waits for the reply. Why doesn't the client send multiple commands at once, for example, a single write that contains the HELO, MAIL, RCPT, DATA, and QUIT commands (assuming the body isn't too large)?

28.4 How can this half-duplex operation of SMTP fool the slow start mechanism when the network is running near capacity?

28.5 When multiple MX records exist with the same preference value, should they always be returned by a name server in the same order?

29

NFS: Network File System

29.1 Introduction

In this chapter we describe NFS, the Network File System, another popular application that provides transparent file access for client applications. The building block of NFS is Sun RPC: Remote Procedure Call, which we must describe first.

Nothing special need be done by the client program to use NFS. The kernel detects that the file being accessed is on an NFS server and automatically generates the RPC calls to access the file.

Our interest in NFS is not in all the details on file access, but in its use of the Internet protocols, especially UDP.

29.2 Sun Remote Procedure Call

Most network programming is done by writing application programs that call system-provided functions to perform specific network operations. For example, one function performs a TCP active open, another performs a TCP passive open, another sends data across a TCP connection, another sets specific protocol options (enable TCP's keepalive timer), and so on. In Section 1.15 we mentioned that two popular sets of functions for network programming (called APIs) are sockets and TLI. The API used by the client and the API used by the server can be different, as can the operating systems running on the client and server. It is the communication protocol and application protocol that determine if a given client and server can communicate with each other. A Unix client written in C using sockets and TCP can communicate with a mainframe server written in COBOL using some other API and TCP, if both hosts are connected across a network and both have a TCP/IP implementation.

Typically the client sends commands to the server, and the server sends replies back to the client. All the applications we've looked at so far—Ping, Traceroute, routing daemons, and the clients and servers for the DNS, TFTP, BOOTP, SNMP, Telnet, FTP, and SMTP—are built this way.

RPC, *Remote Procedure Call*, is a different way of doing network programming. A client program is written that just calls functions in the server program. This is how it appears to the programmer, but the following steps actually take place.

1. When the client calls the remote procedure, it's really calling a function on the local host that's generated by the RPC package. This function is called the *client stub*. The client stub packages the procedure arguments into a network message, and sends this message to the server.
2. A *server stub* on the server host receives the network message. It takes the arguments from the network message, and calls the server procedure that the application programmer wrote.
3. When the server function returns, it returns to the server stub, which takes the return values, packages them into a network message, and sends the message back to the client stub.
4. The client stub takes the return values from the network message and returns to the client application.

The network programming done by the stubs and the RPC library routines uses an API such as sockets or TLI, but the user application—the client program, and the server procedures called by the client—never deal with this API. The client application just calls the server procedures and all the network programming details are hidden by the RPC package, the client stub, and the server stub.

An RPC package provides numerous benefits.

1. The programming is easier since there is little or no network programming involved. The application programmer just writes a client program and the server procedures that the client calls.
2. If an unreliable protocol such as UDP is used, details like timeout and retransmission are handled by the RPC package. This simplifies the user application.
3. The RPC library handles any required data translation for the arguments and return values. For example, if the arguments consist of integers and floating point numbers, the RPC package handles any differences in the way integers and floating point numbers are stored on the client and server. This simplifies coding clients and servers that can operate in heterogeneous environments.

Details of RPC programming are provided in Chapter 18 of [Stevens 1990]. Two popular RPC packages are Sun RPC and the RPC package in the Open Software Foundation's (OSF) Distributed Computing Environment (DCE). Our interest in RPC is to see what the procedure call and procedure return messages look like for the Sun RPC package, since it's used by the Network File System, which we describe in this chapter. Version 2 of Sun RPC is defined in RFC 1057 [Sun Microsystems 1988a].

Sun RPC

Sun RPC comes in two flavors. One version is built using the sockets API and works with TCP and UDP. Another, called TI-RPC (for "transport independent"), is built using the TLI API and works with any transport layer provided by the kernel. From our perspective the two are the same, although we talk only about TCP and UDP in this chapter.

Figure 29.1 shows the format of an RPC procedure call message, when UDP is used.

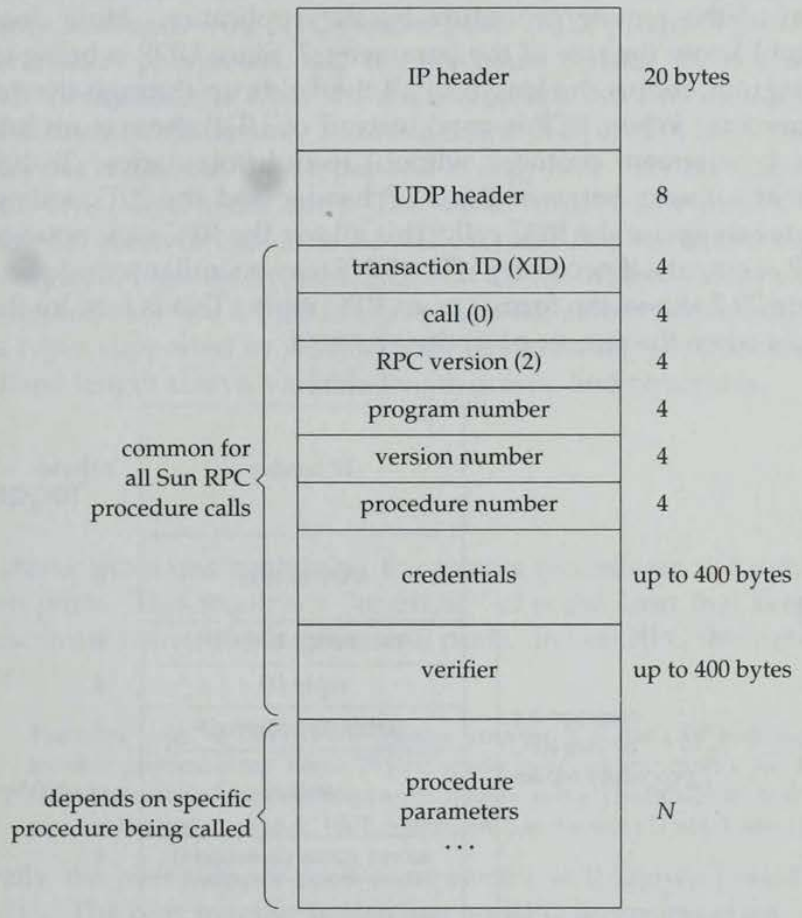


Figure 29.1 Format of RPC procedure call message as a UDP datagram.

The IP and UDP headers are the standard ones we showed earlier (Figures 3.1 and 11.2). What follows after the UDP header is defined by the RPC package.

The *transaction ID* (XID) is set by the client and returned by the server. When the client receives a reply it compares the XID returned by the server with the XID of the request it sent. If they don't match, the client discards the message and waits for the next one from the server. Each time the client issues a new RPC, it changes the XID. But if the client retransmits a previously sent RPC (because it hasn't received a reply), the XID does not change.

The *call* variable is 0 for a call, and 1 for a reply. The current *RPC version* is 2. The next three variables, *program number*, *version number*, and *procedure number*, identify the specific procedure on the server to be called.

The *credentials* identify the client. In some instances nothing is sent here, and in other instances the numeric user ID and group IDs of the client are sent. The server can look at the credentials and determine if it will perform the request or not. The *verifier* is used with Secure RPC, which uses DES encryption. Although the credentials and verifier are variable-length fields, their length is encoded as part of the field.

Following this are the procedure parameters. The format of these depends on the definition of the remote procedure by the application. How does the receiver (the server stub) know the size of the parameters? Since UDP is being used, the size of the UDP datagram, minus the length of all the fields up through the verifier, is the size of the parameters. When TCP is used instead of UDP, there is no inherent length, since TCP is a byte stream protocol, without record boundaries. To handle this, a 4-byte length field appears between the TCP header and the XID, telling the receiver how many bytes comprise the RPC call. This allows the RPC call message to be sent in multiple TCP segments, if necessary. (The DNS uses a similar technique; see Exercise 14.4.)

Figure 29.2 shows the format of an RPC reply. This is sent by the server stub to the client stub, when the remote procedure returns.

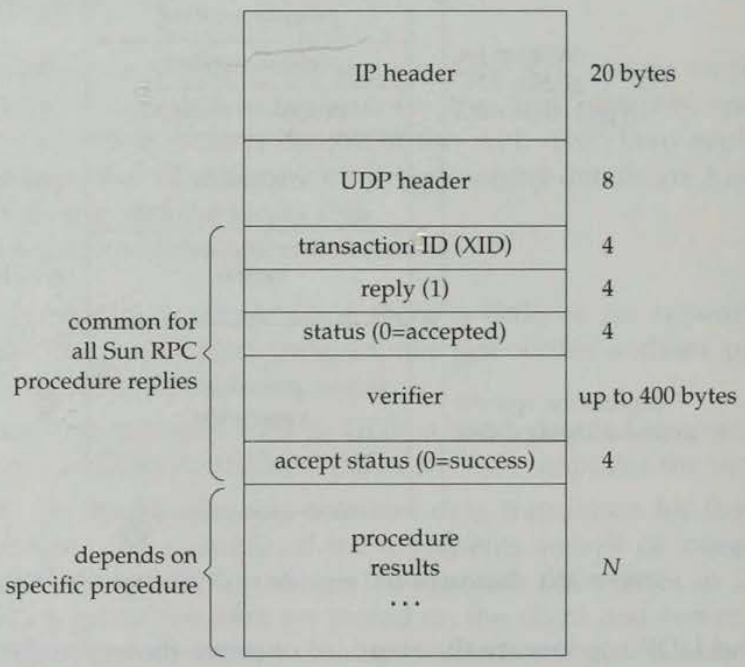


Figure 29.2 Format of RPC procedure reply message as a UDP datagram.

The XID in the reply is just copied from the XID in the call. The *reply* is 1, which we said differentiates this message from a call. The *status* is 0 if the call message was accepted. (The message can be rejected if the RPC version number isn't 2, or if the server cannot authenticate the client.) The *verifier* is used with secure RPC to identify the server.

The *accept status* is 0 on success. A nonzero value can indicate an invalid version number or an invalid procedure number, for example. As with the RPC call message, if TCP is used instead of UDP, a 4-byte length field is sent between the TCP header and the XID.

29.3 XDR: External Data Representation

XDR, *External Data Representation*, is the standard used to encode the values in the RPC call and reply messages—the RPC header fields (XID, program number, accept status, etc.), the procedure parameters, and the procedure results. Having a standard way of encoding all these values is what lets a client on one system call a procedure on a system with a different architecture. XDR is defined in RFC 1014 [Sun Microsystems 1987].

XDR defines numerous data types and exactly how they are transmitted in an RPC message (bit order, byte order, etc.). The sender must build an RPC message in XDR format, then the receiver converts the XDR format into its native representation. We see, for example, in Figures 29.1 and 29.2, that all the integer values we show (XID, call, program number, etc.) are 4-byte integers. Indeed, all integers occupy 4 bytes in XDR. Other data types supported by XDR include unsigned integers, booleans, floating point numbers, fixed-length arrays, variable-length arrays, and structures.

29.4 Port Mapper

The RPC server programs containing the remote procedures use ephemeral ports, not well-known ports. This requires a "registrar" of some form that keeps track of which RPC programs are using which ephemeral ports. In Sun RPC this registrar is called the *port mapper*.

> The term "port" in this name originates from the TCP and UDP port numbers, features of the Internet protocol suite. Since TI-RPC works over any transport layer, and not just TCP and UDP, the name of the port mapper in systems using TI-RPC (SVR4 and Solaris 2.2, for example) has become `rpcbind`. We'll continue to use the more familiar name of port mapper.

Naturally, the port mapper itself must have a well-known port: UDP port 111 and TCP port 111. The port mapper is also just an RPC server program. It has a program number (100000), a version number (2), a TCP port of 111, and a UDP port of 111. Servers register themselves with the port mapper using RPC calls, and clients query the port mapper using RPC calls. The port mapper provides four server procedures:

1. PMAPPROC_SET. Called by an RPC server on startup to register a program number, version number, and protocol with a port number.
2. PMAPPROC_UNSET. Called by server to remove a previously registered mapping.
3. PMAPPROC_GETPORT. Called by an RPC client on startup to obtain the port number for a given program number, version number, and protocol.

4. PMAPPROC_DUMP. Returns all entries (program number, version number, protocol, and port number) in the port mapper database.

When an RPC server program starts, and is later called by an RPC client program, the following steps take place.

1. The port mapper must be started first, normally when the system is bootstrapped. It creates a TCP end point and does a passive open on TCP port 111. It also creates a UDP end point and waits for a UDP datagram to arrive for UDP port 111.
2. When the RPC server program starts, it creates a TCP end point and a UDP end point for each version of the program that it supports. (A given RPC program can support multiple versions. The client specifies which version it wants when it calls a server procedure.) An ephemeral port number is bound to both end points. (It doesn't matter whether the TCP port number is the same or different from the UDP port number.) The server registers each program, version, protocol, and port number by making a remote procedure call to the port mapper's PMAPPROC_SET procedure.
3. When the RPC client program starts, it calls the port mapper's PMAPPROC_GETPORT procedure to obtain the ephemeral port number for a given program, version, and protocol.
4. The client sends an RPC call message to the port number returned in step 3. If UDP is being used, the client just sends a UDP datagram containing an RPC call message (Figure 29.1) to the server's UDP port number. The server responds by sending a UDP datagram containing an RPC reply message (Figure 29.2) back to the client.

   If TCP is being used, the client does an active open to the server's TCP port number, and then sends an RPC call message across the connection. The server responds with an RPC reply message across the connection.

The program `rpcinfo`(8) prints out the port mapper's current mappings. (It calls the port mapper's PMAPPROC_DUMP procedure.) Here is some typical output:

```
sun % /usr/etc/rpcinfo -p
   program vers proto   port
    100005    1   tcp    702  mountd      mount daemon for NFS
    100005    1   udp    699  mountd
    100005    2   tcp    702  mountd
    100005    2   udp    699  mountd
    100003    2   udp   2049  nfs         NFS itself
    100021    1   tcp    709  nlockmgr    NFS lock manager
    100021    1   udp   1036  nlockmgr
    100021    2   tcp    721  nlockmgr
    100021    2   udp   1039  nlockmgr
    100021    3   tcp    713  nlockmgr
    100021    3   udp   1037  nlockmgr
```

We see that some programs do support multiple versions, and each combination of a program number, version number, and protocol has its own port number mapping maintained by the port mapper.

Both versions of the mount daemon are accessed through the same TCP port number (702) and the same UDP port number (699), but each version of the lock manager has its own port number.

29.5 NFS Protocol

NFS provides transparent *file access* for clients to files and filesystems on a server. This differs from FTP (Chapter 27), which provides *file transfer*. With FTP a complete copy of the file is made. NFS accesses only the portions of a file that a process references, and a goal of NFS is to make this access *transparent*. This means that any client application that works with a local file should work with an NFS file, without any program changes whatsoever.

NFS is a client–server application built using Sun RPC. NFS clients access files on an NFS server by sending RPC requests to the server. While this could be done using normal user processes—that is, the NFS client could be a user process that makes explicit RPC calls to the server, and the server could also be a user process—NFS is normally not implemented this way for two reasons. First, accessing an NFS file must be transparent to the client. Therefore the NFS client calls are performed by the client operating system, on behalf of client user processes. Second, NFS servers are implemented within the operating system on the server for efficiency. If the NFS server were a user process, every client request and server reply (including the data being read or written) would have to cross the boundary between the kernel and the user process, which is expensive.

In this section we look at version 2 of NFS, as documented in RFC 1094 [Sun Microsystems 1988b]. A better description of Sun RPC, XDR, and NFS is given in [X/Open 1991]. Details on using and administering NFS are in [Stern 1991]. The specifications for version 3 of the NFS protocol were released in 1993, which we cover in Section 29.7.

Figure 29.3 shows the typical arrangement of an NFS client and an NFS server. There are many subtle points in this figure.

1. It is transparent to the client whether it's accessing a local file or an NFS file. The kernel determines this when the file is opened. After the file is opened, the kernel passes all references to local files to the box labeled "local file access," and all references to an NFS file are passed to the "NFS client" box.
2. The NFS client sends RPC requests to the NFS server through its TCP/IP module. NFS is used predominantly with UDP, but newer implementations can also use TCP.
3. The NFS server receives client requests as UDP datagrams on port 2049. Although NFS could be made to use the port mapper, allowing the server to use an ephemeral port, UDP port 2049 is hardcoded into most implementations.

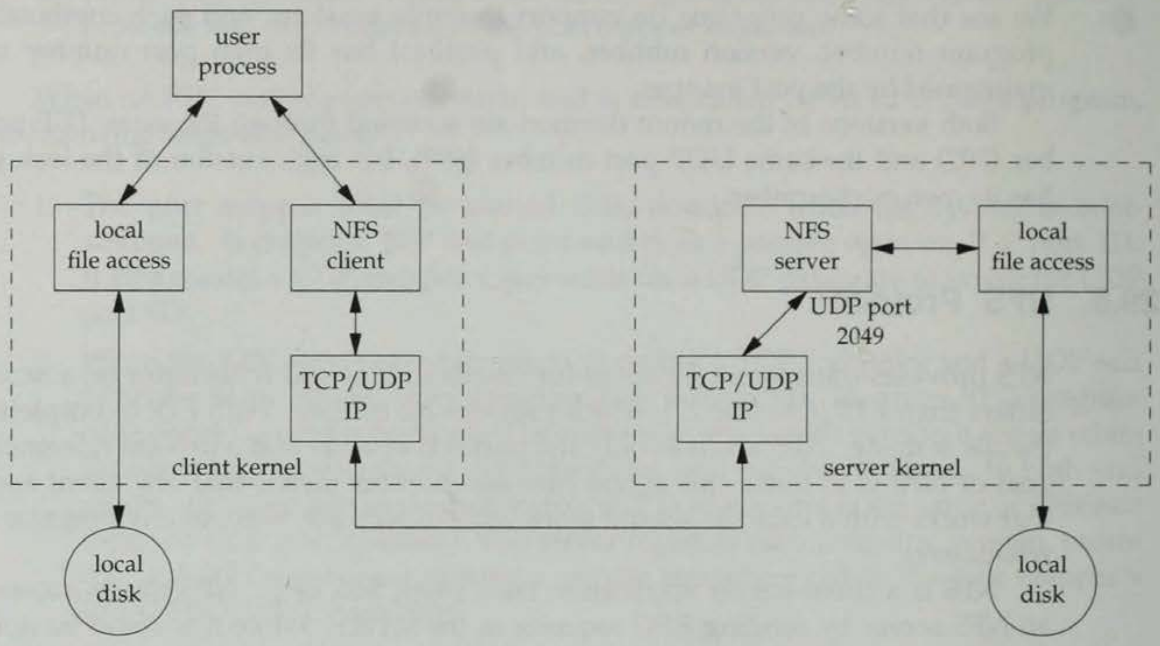


Figure 29.3 Typical arrangement of NFS client and NFS server.

4. When the NFS server receives a client request, the requests are passed to its local file access routines, which access a local disk on the server.
5. It can take the NFS server a while to handle a client's request. The local file-system is normally accessed, which can take some time. During this time, the server does not want to block other client requests from being serviced. To handle this, most NFS servers are multithreaded—that is, there are really multiple NFS servers running inside the server kernel. How this is handled depends on the operating system. Since most Unix kernels are not multithreaded, a common technique is to start multiple instances of a user process (often called `nfsd`) that performs a single system call and remains inside the kernel as a kernel process.
6. Similarly, it can take the NFS client a while to handle a request from a user process on the client host. An RPC is issued to the server host, and the reply is waited for. To provide more concurrency to the user processes on the client host that are using NFS, there are normally multiple NFS clients running inside the client kernel. Again, the implementation depends on the operating system. Unix systems often use a technique similar to the NFS server technique: a user process named `biod` that performs a single system call and remains inside the kernel as a kernel process.

Most Unix hosts can operate as either an NFS client, an NFS server, or both. Most PC implementations (MS-DOS) only provide NFS client implementations. Most IBM mainframe implementations only provide NFS server functions.

NFS really consists of more than just the NFS protocol. Figure 29.4 shows the various RPC programs normally used with NFS.

Application	Program number	Version numbers	Number of procedures
port mapper	100000	2	4
NFS	100003	2	15
mount	100005	1	5
lock manager	100021	1, 2, 3	19
status monitor	100024	1	6

Figure 29.4 Various RPC programs used with NFS.

The versions we show in this figure are the ones found on systems such as SunOS 4.1.3. Newer implementations are providing newer versions of some of the programs. Solaris 2.2, for example, also supports versions 3 and 4 of the port mapper, and version 2 of the mount daemon. SVR4 also supports version 3 of the port mapper.

The mount daemon is called by the NFS client host before the client can access a filesystem on the server. We discuss this below.

The lock manager and status monitor allow clients to lock portions of files that reside on an NFS server. These two programs are independent of the NFS protocol because locking requires state on both the client and server, and NFS itself is stateless on the server. (We say more about NFS's statelessness later.) Chapters 9, 10, and 11 of [X/Open 1991] document the procedures used by the lock manager and status monitor for file locking with NFS.

File Handles

A fundamental concept in NFS is the *file handle*. It is an *opaque* object used to reference a file or directory on the server. The term opaque denotes that the server creates the file handle, passes it back to the client, and then the client uses the file handle when accessing the file. The client never looks at the contents of the file handle—its contents only make sense to the server.

Each time a client process opens a file that is really a file on an NFS server, the NFS client obtains a file handle for that file from the NFS server. Each time the NFS client reads or writes that file for the user process, the file handle is sent back to the server to identify the file being accessed.

Normal user processes never deal with file handles—it is the NFS client code and the NFS server code that pass them back and forth. In version 2 of NFS a file handle occupies 32 bytes, although this increases with version 3 to 64 bytes.

Unix servers normally store the following information in the file handle: the filesystem identifier (the major and minor device numbers of the filesystem), the i-node number (a unique number within a filesystem), and an i-node generation number (a number that changes each time an i-node is reused for a different file).

Mount Protocol

The client must use the NFS mount protocol to mount a server's filesystem, before the client can access files on that filesystem. This is normally done when the client is bootstrapped. The end result is for the client to obtain a file handle for the server's filesystem.

Figure 29.5 shows the sequence of steps that takes place when a Unix client issues the `mount`(8) command, specifying an NFS mount.

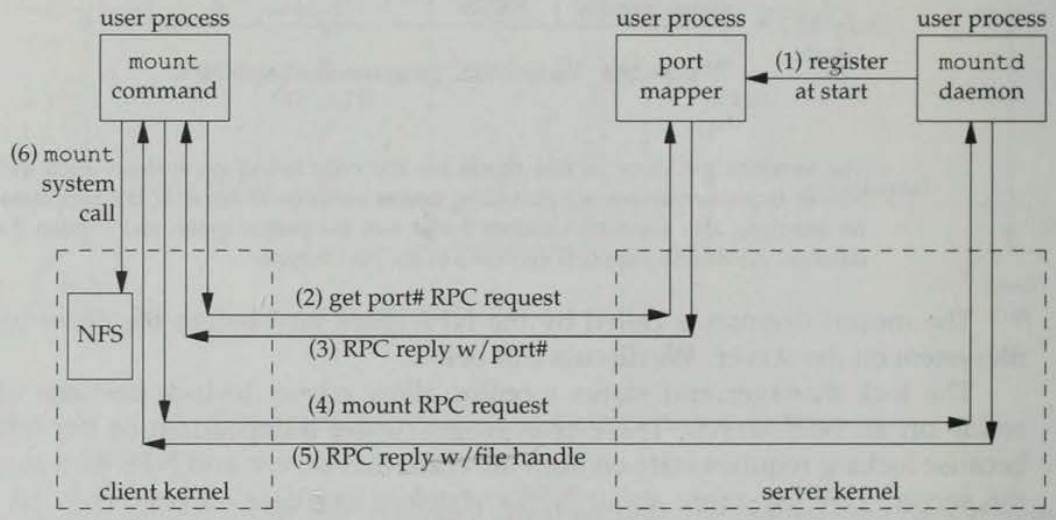


Figure 29.5 Mount protocol used by Unix `mount` command.

The following steps take place.

0. The port mapper is started on the server, normally when the server bootstraps.
1. The mount daemon (`mountd`) is started on the server, after the port mapper. It creates a TCP end point and a UDP end point, and assigns ephemeral port number to each. It then registers these port numbers with the port mapper.
2. The `mount` command is executed on the client and it issues an RPC call to the port mapper on the server to obtain the port number of the server's mount daemon. Either TCP or UDP can be used for this client exchange with the port mapper, but UDP is normally used.
3. The port mapper replies with the port number.
4. The `mount` command issues an RPC call to the mount daemon to mount a filesystem on the server. Again, either TCP or UDP can be used, but UDP is typical. The server can now validate the client, using the client's IP address and port number, to see if the server lets this client mount the specified filesystem.
5. The mount daemon replies with the file handle for the given filesystem.
6. The `mount` command issues the `mount` system call on the client to associate the file handle returned in step 5 with a local mount point on the client. This file

handle is stored in the NFS client code, and from this point on any references by user processes to files on that server's filesystem will use that file handle as the starting point.

This implementation technique puts all the mount processing, other than the `mount` system call on the client, in user processes, instead of the kernel. The three programs we show—the `mount` command, the port mapper, and the mount daemon—are all user processes.

As an example, on our host `sun` (the NFS client) we execute

```
sun # mount -t nfs bsdi:/usr /nfs/bsdi/usr
```

This mounts the directory `/usr` on the host `bsdi` (the NFS server) as the local filesystem `/nfs/bsdi/usr`. Figure 29.6 shows the result.

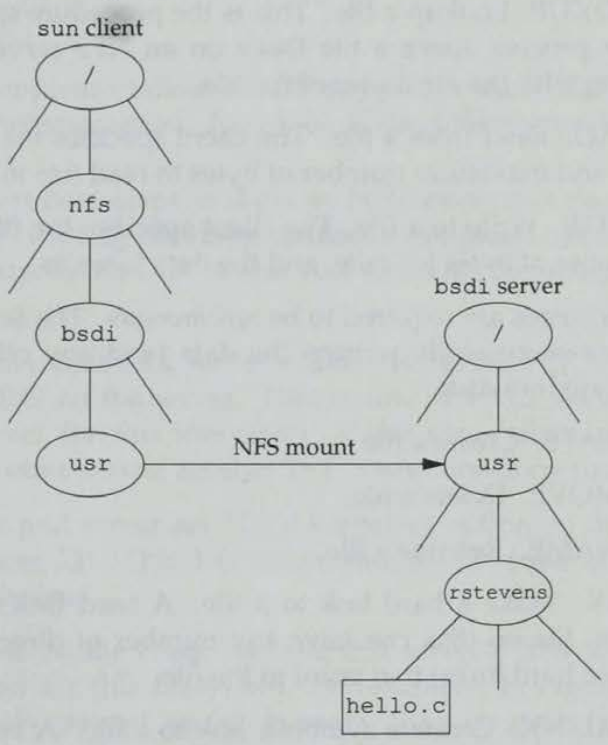


Figure 29.6 Mounting the `bsdi:/usr` directory as `/nfs/bsdi/usr` on the host `sun`.

When we reference the file `/nfs/bsdi/usr/rstevens/hello.c` on the client `sun` we are really referencing the file `/usr/rstevens/hello.c` on the server `bsdi`.

NFS Procedures

The NFS server provides 15 procedures, which we now describe. (The numbers we use are not the same as the NFS procedure numbers, since we have grouped them according to functionality.) Although NFS was designed to work between different operating systems, and not just Unix systems, some of the procedures provide Unix functionality that

might not be supported by other operating systems (e.g., hard links, symbolic links, group owner, execute permission, etc.). Chapter 4 of [Stevens 1992] contains additional information on the properties of Unix filesystems, some of which are assumed by NFS.

1. GETATTR. Return the attributes of a file: type of file (regular file, directory, etc.), permissions, size of file, owner of file, last-access time, and so on.
2. SETATTR. Set the attributes of a file. Only a subset of the attributes can be set: permissions, owner, group owner, size, last-access time, and last-modification time.
3. STATFS. Return the status of a filesystem: amount of available space, optimal size for transfer, and so on. Used by the Unix `df` command, for example.
4. LOOKUP. Lookup a file. This is the procedure called by the client each time a user process opens a file that's on an NFS server. A file handle is returned, along with the attributes of the file.
5. READ. Read from a file. The client specifies the file handle, starting byte offset, and maximum number of bytes to read (up to 8192).
6. WRITE. Write to a file. The client specifies the file handle, starting byte offset, number of bytes to write, and the data to write.

   NFS writes are required to be *synchronous*. The server cannot respond OK until it has successfully written the data (and any other file information that gets updated) to disk.
7. CREATE. Create a file.
8. REMOVE. Delete a file.
9. RENAME. Rename a file.
10. LINK. Make a hard link to a file. A hard link is a Unix concept whereby a given file on disk can have any number of directory entries (i.e., names, also called hard links) that point to the file.
11. SYMLINK. Create a symbolic link to a file. A symbolic link is a file that contains the name of another file. Most operations that reference the symbolic link (e.g., open) really reference the file pointed to by the symbolic link.
12. READLINK. Read a symbolic link, that is, return the name of the file to which the symbolic link points.
13. MKDIR. Create a directory.
14. RMDIR. Delete a directory.
15. READDIR. Read a directory. Used by the Unix `ls` command, for example.

These procedure names actually begin with the prefix NFSPROC_, which we've dropped.

UDP or TCP?

NFS was originally written to use UDP, and that's what all vendors provide. Newer implementations, however, also support TCP. TCP support is provided for use on wide area networks, which are getting faster over time. NFS is no longer restricted to local area use.

The network dynamics can change drastically when going from a LAN to a WAN. The round-trip times can vary widely and congestion is more frequent. These characteristics of WANs led to the algorithms we examined with TCP—slow start and congestion avoidance. Since UDP does not provide anything like these algorithms, either the same algorithms must be put into the NFS client and server or TCP should be used.

NFS Over TCP

The Berkeley Net/2 implementation of NFS supports either UDP or TCP. [Macklem 1991] describes this implementation. Let's look at the differences when TCP is used.

1. When the server bootstraps, it starts an NFS server that does a passive open on TCP port 2049, waiting for client connection requests. This is usually in addition to the normal NFS UDP server that waits for incoming datagrams to UDP port 2049.
2. When the client mounts the server's filesystem using TCP, it does an active open to TCP port 2049 on the server. This results in a TCP connection between the client and server for this filesystem. If the same client mounts another filesystem on the same server, another TCP connection is created.
3. Both the client and server set TCP's keepalive option on their ends of the connection (Chapter 23). This lets either end detect if the other end crashes, or crashes and reboots.
4. All applications on the client that use this server's filesystem share the single TCP connection for this filesystem. For example, in Figure 29.6 if there were another directory named `smith` beneath `/usr` on `bsdi`, references to files in `/nfs/bsdi/usr/rstevens` and `/nfs/bsdi/usr/smith` would share the same TCP connection.
5. If the client detects that the server has crashed, or crashed and rebooted (by receiving a TCP error of either "connection timed out" or "connection reset by peer"), it tries to reconnect to the server. The client does another active open to reestablish the TCP connection with the server for this filesystem. Any client requests that timed out on the previous connection are reissued on the new connection.
6. If the client crashes, so do the applications that are running when it crashes. When the client reboots, it will probably remount the server's filesystem using TCP, resulting in another TCP connection to the server. The previous connection

between this client and server for this filesystem is half-open (the server thinks it's still open), but since the server set the keepalive option, this half-open connection will be terminated when the next keepalive probe is sent by the server's TCP.

Over time, additional vendors plan to support NFS over TCP.

29.6 NFS Examples

Let's use `tcpdump` to see which NFS procedures are invoked by the client for typical file operations. When `tcpdump` detects a UDP datagram containing an RPC call (*call* equals 0 in Figure 29.1) with a destination port of 2049, it decodes the datagram as an NFS request. Similarly if the UDP datagram is an RPC reply (*reply* equals 1 in Figure 29.2) with a source port of 2049, it decodes the datagram as an NFS reply.

Simple Example: Reading a File

Our first example just copies a file to the terminal using the `cat`(1) command, but the file is on an NFS server:

```
sun % cat /nfs/bsdi/usr/rstevens/hello.c      copy file to terminal
main()
{
        printf("hello, world\n");
}
```

On the host `sun` (the NFS client) the filesystem `/nfs/bsdi/usr` is really the `/usr` filesystem on the host `bsdi` (the NFS server), as shown in Figure 29.6. The kernel on `sun` detects this when `cat` opens the file, and uses NFS to access the file. Figure 29.7 shows the `tcpdump` output.

```
 1  0.0                  sun.7aa6 > bsdi.nfs: 104 getattr
 2  0.003587 (0.0036)    bsdi.nfs > sun.7aa6: reply ok 96

 3  0.005390 (0.0018)    sun.7aa7 > bsdi.nfs: 116 lookup "rstevens"
 4  0.009570 (0.0042)    bsdi.nfs > sun.7aa7: reply ok 128

 5  0.011413 (0.0018)    sun.7aa8 > bsdi.nfs: 116 lookup "hello.c"
 6  0.015512 (0.0041)    bsdi.nfs > sun.7aa8: reply ok 128

 7  0.018843 (0.0033)    sun.7aa9 > bsdi.nfs: 104 getattr
 8  0.022377 (0.0035)    bsdi.nfs > sun.7aa9: reply ok 96

 9  0.027621 (0.0052)    sun.7aaa > bsdi.nfs: 116 read 1024 bytes @ 0
10  0.032170 (0.0045)    bsdi.nfs > sun.7aaa: reply ok 140
```

Figure 29.7 NFS operations to read a file.

When `tcpdump` decodes an NFS request or reply, it prints the XID field for the client, instead of the port number. The XID field in lines 1 and 2 is `0x7aa6`.

The filename `/nfs/bsdi/usr/rstevens/hello.c` is processed by the open function in the client kernel one element at a time. When it reaches `/nfs/bsdi/usr` it detects that this is a mount point to an NFS mounted filesystem.

In line 1 the client calls the GETATTR procedure to fetch the attributes of the server's directory that the client has mounted (`/usr`). This RPC request contains 104 bytes of data, exclusive of the IP and UDP headers. The reply in line 2 has a return value of OK and contains 96 bytes of data, exclusive of the IP and UDP headers. We see in this figure that the minimum NFS message contains around 100 bytes of data.

In line 3 the client calls the LOOKUP procedure for the file `rstevens` and receives an OK reply in line 4. The LOOKUP specifies the filename `rstevens` and the file handle that was saved by the kernel when the remote filesystem was mounted. The reply contains a new file handle that is used in the next step.

In line 5 the client does a LOOKUP of `hello.c` using the file handle from line 4. It receives another file handle in line 6. This new file handle is what the client uses in lines 7 and 9 to reference the file `/nfs/bsdi/usr/rstevens/hello.c`. We see that the client does a LOOKUP for each component of the pathname that is being opened.

In line 7 the client does another GETATTR, followed by a READ in line 9. The client asks for 1024 bytes, starting at offset 0, but receives less. (After subtracting the sizes of the RPC fields, and the other values returned by the READ procedure, 38 bytes of data are returned in line 10. This is indeed the size of the file `hello.c`.)

In this example the user process knows nothing about these NFS requests and replies that are being done by the kernel. The application just calls the kernel's open function, which causes 3 requests and 3 replies to be exchanged (lines 1–6), and then calls the kernel's read function, which causes 2 requests and 2 replies (lines 7–10). It is transparent to the client application that the file is on an NFS server.

Simple Example: Creating a Directory

As another simple example we'll change our working directory to a directory that's on an NFS server, and then create a new directory:

```
sun % cd /nfs/bsdi/usr/rstevens          change working directory
sun % mkdir Mail                         and create a directory
```

Figure 29.8 shows the `tcpdump` output.

```
 1   0.0                  sun.7ad2 > bsdi.nfs: 104 getattr
 2   0.004912 ( 0.0049)   bsdi.nfs > sun.7ad2: reply ok 96

 3   0.007266 ( 0.0024)   sun.7ad3 > bsdi.nfs: 104 getattr
 4   0.010846 ( 0.0036)   bsdi.nfs > sun.7ad3: reply ok 96

 5  35.769875 (35.7590)   sun.7ad4 > bsdi.nfs: 104 getattr
 6  35.773432 ( 0.0036)   bsdi.nfs > sun.7ad4: reply ok 96

 7  35.775236 ( 0.0018)   sun.7ad5 > bsdi.nfs: 112 lookup "Mail"
 8  35.780914 ( 0.0057)   bsdi.nfs > sun.7ad5: reply ok 28

 9  35.782339 ( 0.0014)   sun.7ad6 > bsdi.nfs: 144 mkdir "Mail"
10  35.992354 ( 0.2100)   bsdi.nfs > sun.7ad6: reply ok 128
```

Figure 29.8 NFS operations for `cd` to NFS directory, then `mkdir`.

Changing our directory causes the client to call the GETATTR procedure twice (lines 1–4). When we create the new directory, the client calls the GETATTR procedure (lines 5 and 6), followed by a LOOKUP (lines 7 and 8, to verify that the directory doesn't already exist), followed by a MKDIR to create the directory (lines 9 and 10). The reply of OK in line 8 doesn't mean that the directory exists. It just means the procedure returned. `tcpdump` doesn't interpret the return values from the NFS procedures. It normally prints OK and the number of bytes of data in the reply.

Statelessness

One of the features of NFS (critics of NFS would call this a wart, not a feature) is that the NFS server is stateless. The server does not keep track of which clients are accessing which files. Notice in the list of NFS procedures shown earlier that there is not an open procedure or a close procedure. The LOOKUP procedure is similar to an open, but the server never knows if the client is really going to reference the file after the client does a LOOKUP.

The reason for a stateless design is to simplify the crash recovery of the server after it crashes and reboots.

Example: Server Crash

In the following example we are reading a file from an NFS server when the server crashes and reboots. This shows how the stateless server approach lets the client "not know" that the server crashes. Other than a time pause while the server crashes and reboots, the client is unaware of the problem, and the client application is not affected.

On the client `sun` we start a `cat` of a long file (`/usr/share/lib/termcap` on the NFS server `svr4`), disconnect the Ethernet cable during the transfer, shut down and reboot the server, then reconnect the cable. The client was configured to read 1024 bytes per NFS read. Figure 29.9 shows the `tcpdump` output.

Lines 1–10 correspond to the client opening the file. The operations are similar to those shown in Figure 29.7. In line 11 we see the first READ of the file, with 1024 bytes of data returned in line 12. This continues (a READ of 1024 followed by a reply of OK) through line 129.

In lines 130 and 131 we see two requests that time out and are retransmitted in lines 132 and 133. The first question is why are there two read requests, one starting at offset 65536 and the other starting at 73728? The client kernel has detected that the client application is performing sequential reads, and is trying to prefetch data blocks. (Most Unix kernels do this *read-ahead*.) The client kernel is also running multiple NFS block I/O daemons (`biod` processes) that try to generate multiple RPC requests on behalf of clients. One daemon is reading 8192 bytes starting at 65536 (in 1024-byte chunks) and the other is performing the read-ahead of 8192 bytes starting at 73728.

Client retransmissions occur in lines 130–168. In line 169 we see the server has rebooted, and it sends an ARP request before it can reply to the client's NFS request in line 168. The response to line 168 is sent in line 171. The client READ requests continue.

```
 1   0.0                 sun.7ade > svr4.nfs: 104 getattr
 2   0.007653 ( 0.0077)  svr4.nfs > sun.7ade: reply ok 96

 3   0.009041 ( 0.0014)  sun.7adf > svr4.nfs: 116 lookup "share"
 4   0.017237 ( 0.0082)  svr4.nfs > sun.7adf: reply ok 128

 5   0.018518 ( 0.0013)  sun.7ae0 > svr4.nfs: 112 lookup "lib"
 6   0.026802 ( 0.0083)  svr4.nfs > sun.7ae0: reply ok 128

 7   0.028096 ( 0.0013)  sun.7ae1 > svr4.nfs: 116 lookup "termcap"
 8   0.036434 ( 0.0083)  svr4.nfs > sun.7ae1: reply ok 128

 9   0.038060 ( 0.0016)  sun.7ae2 > svr4.nfs: 104 getattr
10   0.045821 ( 0.0078)  svr4.nfs > sun.7ae2: reply ok 96

11   0.050984 ( 0.0052)  sun.7ae3 > svr4.nfs: 116 read 1024 bytes @ 0
12   0.084995 ( 0.0340)  svr4.nfs > sun.7ae3: reply ok 1124
                         reading continues

128   3.430313 ( 0.0013)  sun.7b22 > svr4.nfs: 116 read 1024 bytes @ 64512
129   3.441828 ( 0.0115)  svr4.nfs > sun.7b22: reply ok 1124

130   4.125031 ( 0.6832)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
131   4.868593 ( 0.7436)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

132   4.993021 ( 0.1244)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
133   5.732217 ( 0.7392)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

134   6.732084 ( 0.9999)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
135   7.472098 ( 0.7400)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

136  10.211964 ( 2.7399)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
137  10.951960 ( 0.7400)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

138  17.171767 ( 6.2198)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
139  17.911762 ( 0.7400)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

140  31.092136 (13.1804)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
141  31.831432 ( 0.7393)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

142  51.090854 (19.2594)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
143  51.830939 ( 0.7401)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728

144  71.090305 (19.2594)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
145  71.830155 ( 0.7398)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
                         retransmissions continue

167 291.824285 ( 0.7400)  sun.7b24 > svr4.nfs: 116 read 1024 bytes @ 73728
168 311.083676 (19.2594)  sun.7b23 > svr4.nfs: 116 read 1024 bytes @ 65536
                         server reboots

169 311.149476 ( 0.0658)  arp who-has sun tell svr4
170 311.150004 ( 0.0005)  arp reply sun is-at 8:0:20:3:f6:42

171 311.154852 ( 0.0048)  svr4.nfs > sun.7b23: reply ok 1124

172 311.156671 ( 0.0018)  sun.7b25 > svr4.nfs: 116 read 1024 bytes @ 66560
173 311.168926 ( 0.0123)  svr4.nfs > sun.7b25: reply ok 1124
                         reading continues
```

Figure 29.9 Client reading a file when an NFS server crashes and reboots.

The client application never knows that the server crashes and reboots, and except for the 5-minute pause between lines 129 and 171, this server crash is transparent to the client.

To examine the timeout and retransmission interval in this example, realize that there are two client daemons with their own timeouts. The intervals for the first daemon (reading at offset 65536), rounded to two decimal points, are: 0.68, 0.87, 1.74, 3.48, 6.96, 13.92, 20.0, 20.0, 20.0, and so on. The intervals for the second daemon (reading at offset 73728) are the same (to two decimal points). It appears that these NFS clients are using a timeout that is a multiple of 0.875 seconds with an upper bound of 20 seconds. After each timeout the retransmission interval is doubled: 0.875, 1.75, 3.5, 7.0, and 14.0.

How long does the client retransmit? The client has two options that affect this. First, if the server filesystem is mounted *hard*, the client retransmits forever, but if the server filesystem is mounted *soft*, the client gives up after a fixed number of retransmissions. Also, with a hard mount the client has an option of whether to let the user interrupt the infinite retransmissions or not. If the client host specifies interruptibility when it mounts the server's filesystem, if we don't want to wait 5 minutes for the server to reboot after it crashes, we can type our interrupt key to abort the client application.

Idempotent Procedures

An RPC procedure is called *idempotent* if it can be executed more than once by the server and still return the same result. For example, the NFS read procedure is idempotent. As we saw in Figure 29.9, the client just reissues a given READ call until it gets a response. In our example the reason for the retransmission was that the server had crashed. If the server hasn't crashed, and the RPC reply message is lost (since UDP is unreliable), the client just retransmits and the server performs the same READ again. The same portion of the same file is read again and sent back to the client.

This works because each READ request specifies the starting offset of the read. If there were an NFS procedure asking the server to read the *next N* bytes of a file, this wouldn't work. Unless the server is made stateful (as opposed to stateless), if a reply is lost and the client reissues the READ for the next *N* bytes, the result is different. This is why the NFS READ and WRITE procedures have the client specify the starting offset. The client maintains the state (the current offset of each file), not the server.

Unfortunately, not all filesystem operations are idempotent. For example, consider the following steps: the client NFS issues the REMOVE request to delete a file; the server NFS deletes the file and responds OK; the server's response is lost; the client NFS times out and retransmits the request; the server NFS can't find the file and responds with an error; the client application receives an error saying the file doesn't exist. This error return to the client application is wrong—the file did exist and was deleted.

The NFS operations that are idempotent are: GETATTR, STATFS, LOOKUP, READ, WRITE, READLINK, and READDIR. The procedures that are not idempotent are: CREATE, REMOVE, RENAME, LINK, SYMLINK, MKDIR, and RMDIR. SETATTR is normally idempotent, unless it's being used to truncate a file.

Since lost responses can always happen with UDP, NFS servers need a way to handle the nonidempotent operations. Most servers implement a recent-reply cache in which they store recent replies for the nonidempotent operations. Each time the server receives a request, it first checks this cache, and if a match is found, returns the previous reply instead of calling the NFS procedure again. [Juszczak 1989] provides details on this type of cache.

This concept of idempotent server procedures applies to any UDP-based application, not just NFS. The DNS, for example, provides an idempotent service. A DNS server can execute a resolver's request any number of times with no ill effects (other than wasted network resources).

29.7 NFS Version 3

During 1993 the specifications for version 3 of the NFS protocol were released [Sun Microsystems 1994]. Implementations are expected to become available during 1994.

Here we summarize the major differences between versions 2 and 3. We'll refer to the two as V2 and V3.

1. The file handle in V2 is a fixed-size array of 32 bytes. With V3 it becomes a variable-length array up to 64 bytes. A variable-length array in XDR is encoded with a 4-byte count, followed by the actual bytes. This reduces the size of the file handle on implementations such as Unix that only need about 12 bytes, but allows non-Unix implementations to maintain additional information.
2. V2 limits the number of bytes per READ or WRITE RPC to 8192 bytes. This limit is removed in V3, meaning an implementation over UDP is limited only by the IP datagram size (65535 bytes). This allows larger read and write packets on faster networks.
3. File sizes and the starting byte offsets for the READ and WRITE procedures are extended from 32 to 64 bits, allowing larger file sizes.
4. A file's attributes are returned on every call that affects the attributes. This reduces the number of GETATTR calls required by the client.
5. WRITEs can be asynchronous, instead of the synchronous WRITEs required by V2. This can improve WRITE performance.
6. One procedure was deleted (STATFS) and seven were added: ACCESS (check file access permissions), MKNOD (create a Unix special file), READDIRPLUS (returns names of files in a directory along with their attributes), FSINFO (returns the static information about a filesystem), FSSTAT (returns the dynamic information about a filesystem), PATHCONF (returns the POSIX.1 information about a file), and COMMIT (commit previous asynchronous writes to stable storage).

29.8 Summary

RPC is a way to build a client–server application so that it appears that the client just calls server procedures. All the networking details are hidden in the client and server stubs, which are generated for an application by the RPC package, and in the RPC library routines. We showed the format of the RPC call and reply messages, and mentioned that XDR is used to encode the values, allowing RPC clients and servers to run on machines with different architectures.

One of the most widely used RPC applications is Sun's NFS, a heterogeneous file access protocol that is widely implemented on hosts of all sizes. We looked at NFS and the way that it uses UDP and TCP. Fifteen procedures define the NFS Version 2 protocol.

A client's access to an NFS server starts with the mount protocol, returning a file handle to the client. The client can then access files on the server's filesystem using that file handle. Filenames are looked up on the server one element at a time, returning a new file handle for each element. The end result is a file handle for the file being referenced, which is used in subsequent reads and writes.

NFS tries to make all its procedures idempotent, so that the client can just reissue a request if the response gets lost. We saw an example of this with a client reading a file while the server crashed and rebooted.

Exercises

29.1 In Figure 29.7 we saw that `tcpdump` interpreted the packets as NFS requests and replies, printing the XID. Can `tcpdump` do this for any RPC request or reply?

29.2 On a Unix system, why do you think RPC server programs use ephemeral ports and not well-known ports?

29.3 An RPC client calls two server procedures. The first server procedure takes 5 seconds to execute, and the second procedure takes 1 second to execute. The client has a timeout of 4 seconds. Draw a time line of what's exchanged between the client and server. (Assume it takes no time for messages from the client to the server, and vice versa.)

29.4 What would happen in the example shown in Figure 29.9 if, while the NFS server were down, its Ethernet card were replaced?

29.5 When the server reboots in Figure 29.9, it handles the request starting at byte offset 65536 (lines 168 and 171), and then handles the next request starting at offset 66560 (lines 172 and 173). What happened to the request starting at offset 73728 (line 167)?

29.6 When we described idempotent NFS procedures we gave an example of a REMOVE reply being lost in the network. What happens in this case if TCP is used, instead of UDP?

29.7 If the NFS server used an ephemeral port instead of 2049, what would happen to an NFS client when the server crashes and reboots?

29.8 Reserved port numbers (Section 1.9) are scarce, since there are a maximum of 1023 per host. If an NFS server requires its clients to have reserved ports (which is common) and an NFS client using TCP mounts N filesystems on N different servers, does the client need a different reserved port number for each connection?

30

Other TCP/IP Applications

30.1 Introduction

In this chapter we describe additional TCP/IP applications that many implementations support. Some are simple and easy to cover completely (Finger and Whois), while another is complex (the X Window System). We provide only a brief overview of this complex application, focusing on its use of the TCP/IP protocols.

Additionally we provide an overview of some Internet resource discovery tools. These are tools to help us navigate our way around the Internet, searching for items whose location and exact name we don't know.

30.2 Finger Protocol

The Finger protocol returns information on one or more users on a specified host. It's commonly used to see if someone is currently logged on, or to figure out someone's login name, to send them mail. RFC 1288 [Zimmerman 1991] specifies the protocol.

Many sites do not run a Finger server for two reasons. First, a programming error in an earlier version of the server was one of the entry points used by the infamous Internet worm of 1988. (RFC 1135 [Reynolds 1989] and [Curry 1992] describe the worm in more detail.) Second, the Finger protocol can reveal detailed information on users (login names, phone numbers, when they last logged in, etc.) that many administrators consider private. Section 3 of RFC 1288 details the security aspects of this service.

From a protocol perspective, the Finger server has a well-known port of 79. The client does an active open to this port and sends a one-line query The server processes the query, sends back the output, and closes the connection. The query and response are NVT ASCII, similar to what we saw with FTP and SMTP.

While most Unix users access the Finger server using the `finger`(1) client, we'll start by using the Telnet client to connect directly to the server and see the one-line commands issued by the client. If the client query is an empty line (which in NVT ASCII is transmitted as a CR followed by an LF), it is a request for information on all online users.

```
sun % telnet slip finger
Trying 140.252.13.65 ...                    first three lines are output by Telnet client
Connected to slip.
Escape character is '^]'.
                                            here we type RETURN as the Finger client command
Login    Name               Tty  Idle  Login Time   Office   Office Phone
rstevens Richard Stevens    *co    45  Jul 31 09:13
rstevens Richard Stevens    *c2    45  Aug  5 09:41
Connection closed by foreign host.    output by Telnet client
```

The blank output fields for the office and office phone are taken from optional fields in the user's password file entry (which aren't present in this example).

The server must be the end that does the active close, since a variable amount of information is returned by the server, and the reception of the end-of-file by the client is how the client knows when the output is complete.

When the client request consists of a username, the server responds with information only about that user. Here's another example, with the Telnet client output removed:

```
sun % telnet vangogh.cs.berkeley.edu finger
rstevens                                    this is the client request we type
Login: rstevens                             Name: Richard Stevens
Directory: /a/guest/rstevens                Shell: /bin/csh
Last login Thu Aug  5 09:55 (PDT) on ttyq2 from sun.tuc.noao.edu
Mail forwarded to: rstevens@noao.edu
No Plan.
```

When a system has the Finger service completely disabled, the client's active open will receive an RST from the server, since no process has a passive open on port 79:

```
sun % finger @svr4
[svr4.tuc.noao.edu] connect: Connection refused
```

Some sites provide a server on port 79, but it just outputs information to the client, and doesn't honor any client requests:

```
sun % finger @att.com
[att.com]                   this line output by Finger client; remainder from server
------------------------------------------------------------
There are no user accounts on the AT&T Internet gateway.
To send email to an AT&T employee, send email to their name
separated by periods at att.com. If the employee has an email
address registered in the employee database, they will receive
email - otherwise, you'll receive a non-delivery notice.
For example: John.Q.Public@att.com
```

```
sun % finger clinton@whitehouse.gov
[whitehouse.gov]

        Finger service for arbitrary addresses on whitehouse.gov is not
supported. If you wish to send electronic mail, valid addresses are
"PRESIDENT@WHITEHOUSE.GOV", and "VICE-PRESIDENT@WHITEHOUSE.GOV".
```

Another possibility is for an organization to implement a *firewall gateway*: a router between the organization and the Internet that filters out (i.e., discards) certain IP datagrams. ([Cheswick and Bellovin 1994] discuss firewall gateways in detail.) The firewall gateway can be configured to discard incoming datagrams that are TCP segments for port 79. In this case the Finger client times out after about 75 seconds.

There are additional options for the Finger server, and for the Unix `finger` client. Refer to RFC 1288 and the `finger`(1) manual page for the details.

> RFC 1288 states that vending machines with TCP/IP connections that provide a Finger server should reply to a client request consisting of a blank line with a list of all items currently available. They should reply to a client request consisting of a name with a count or list of available items for that product.

30.3 Whois Protocol

The Whois protocol is another information service. Although any site can provide a Whois server, the one at the InterNIC, `rs.internic.net`, is most commonly used. This server maintains information about all registered DNS domains and many system administrators responsible for systems connected to the Internet. (Another server is provided at `nic.ddn.mil`, but contains information only about the MILNET.) Unfortunately the information can be out of data or incomplete. RFC 954 [Harrenstein, Stahl, and Feinler 1985] documents the Whois service.

From a protocol perspective, the Whois server has a well-known TCP port of 43. It accepts connection requests from clients, and the client sends a one-line query to the server. The server responds with whatever information is available and then closes the connection. The requests and replies are transmitted using NVT ASCII. This is almost identical to the Finger server, although the requests and replies contain different information.

The common Unix client is the `whois`(1) program, although we can use Telnet and type in the commands ourself. The starting place is to send a request consisting of just a question mark, which returns more detailed information on the supported client requests.

> When the NIC moved to the InterNIC in 1993, the site for the Whois server moved from `nic.ddn.mil` to `rs.internic.net`. Many vendors still ship versions of the `whois` client with the name `nic.ddn.mil` built in. You may need to specify the command-line argument `-h rs.internic.net` to contact the correct server.
>
> Alternately, we can Telnet to `rs.internic.net` and login as `whois`.

We'll use the Whois server to track down the author. (We've removed the extraneous Telnet client output.) Our first request is for all names that match "stevens."

```
sun % telnet rs.internic.net whois
stevens                                    this is the client command we type

                                           information on 25 other "stevens" that we omit
Stevens, W. Richard (WRS28)    stevens@kohala.com      +1 602 297 9416

The InterNIC Registration Services Host ONLY contains Internet
Information (Networks, ASN's, Domains, and POC's).
Please use the whois server at nic.ddn.mil for MILNET Information.
```

The three uppercase letters followed by a number in parentheses after the name, `(WRS28)`, are the person's NIC handle. The next query contains an exclamation point and a NIC handle, to fetch more information about this person.

```
sun % telnet rs.internic.net whois
!wrs28                                     client request that we type
Stevens, W. Richard (WRS28)                   stevens@kohala.com
   Kohala Software
   1202 E. Paseo del Zorro
   Tucson, AZ 85718
   +1 602 297 9416

   Record last updated on 11-Jan-91.
```

Lots of additional information about Internet variables can also be queried. For example, the request `net 140.252` returns information about the class B address 140.252.

White Pages

Using the VRFY command of SMTP, along with the Finger protocol and the Whois protocol to locate users on the Internet is similar to using the white pages of a telephone book to find someone's phone number. At the present time ad hoc tools such as these are all that's widely available, but research is under way to improve this type of service.

[Schwartz and Tsirigotis 1991] contains additional information on various white pages services being tried on the Internet. The particular tool, called Netfind, can be accessed by using Telnet to either `bruno.cs.colorado.edu` or `ds.internic.net` and logging in as `netfind`.

RFC 1309 [Weider, Reynolds, and Heker 1992] provides an overview of the OSI directory service, called X.500, and compares and contrasts it with current Internet techniques (Finger and Whois).

30.4 Archie, WAIS, Gopher, Veronica, and WWW

The tools that we described in the previous two sections—Finger, Whois, and a white pages service—are for locating information on people. Other tools exist to locate files and documents, and this section gives an overview of these tools. We only provide an

overview, because examining the details of each tool is beyond the scope of this book. Methods are given for accessing these tools across the Internet, and you are encouraged to do so, to find which tool can help you. Additional tools are continually being developed. [Obraczka, Danzig, and Li 1993] provide an overview of resource discovery services on the Internet.

Archie

Many of the resources used in this text were obtained using anonymous FTP. The problem is finding which FTP site has the program we want. Sometimes we don't even know the exact filename, but we know some keywords that probably appear in the filename.

Archie provides a directory of thousands of FTP servers across the Internet. We can access this directory by logging into an Archie server and searching for files whose name contains a specified regular expression. The output is a list of servers with matching filenames. We then use anonymous FTP to that site to fetch the file.

There are many Archie servers across the world. One starting point is to use Telnet to `ds.internic.net`, login as `archie`, and execute the command `servers`. This provides a list of all the Archie servers, and their location.

WAIS: Wide Area Information Servers

Archie helps us locate filenames that contain keywords, but sometimes we're looking for a file or database that contains a keyword. That is, we want to search for a file that contains a keyword, not a filename containing a keyword.

WAIS knows about hundreds of databases that contain information on both computer-related topics and other general topics. To use WAIS we select the databases to search and specify the keywords. To try WAIS Telnet to `quake.think.com` and login as `wais`.

Gopher

Gopher is a menu-driven front end to other Internet resource services, such as Archie, WAIS, and anonymous FTP. Gopher is one of the easiest to use, since its user interface is the same, regardless of which resource service it's using.

To use Gopher, Telnet into `is.internic.net` and login as `gopher`.

Veronica: Very Easy Rodent-Oriented Netwide Index to Computerized Archives

Just as Archie is an index of anonymous FTP servers, Veronica is an index of titles of Gopher items. A Veronica search typically searches hundreds of Gopher servers.

To access Veronica we must go through a Gopher client. Select the Gopher menu item "Beyond InterNIC: Virtual Treasures of the Internet" and then select Veronica from the next menu.

WWW: World Wide Web

World Wide Web lets us browse a large, worldwide set of services and documents using a tool called *hypertext*. As information is displayed, certain keywords are highlighted, and we can select more information on those keywords.

To access WWW, Telnet to `info.cern.ch`.

30.5 X Window System

The *X Window System*, or just X, is a client–server application that lets multiple clients (applications) use the bit-mapped display managed by a server. The server is the software that manages a display, keyboard, and mouse. The client is an application program that runs on either the same host as the server or on a different host. In the latter case the common form of communication between the client and server is TCP, although other protocols such as DECNET can be used. In some instances the server is a dedicated piece of hardware (an X terminal) that communicates with clients on other hosts. In another instance, a stand-alone workstation, the client and server are on the same host and communicate using interprocess communication on that host, without any network involvement at all. Between these two extremes is a workstation that supports clients on the same host and clients on other hosts.

X requires a reliable, bidirectional stream protocol, such as TCP. (X was not designed for an unreliable protocol such as UDP.) The communication between the client and server consists of 8-bit bytes exchanged across this connection. [Nye 1992] gives the format of the more than 150 messages exchanged between the client and server across their TCP connection.

On a Unix system, when the X client and X server are on the same host, the Unix domain protocols are normally used instead of TCP, because there is less protocol processing than if TCP were used. The Unix domain protocols are a form of interprocess communication that can be used between clients and servers on the same host. Recall in Figure 2.4 (p. 28) that when TCP is used for communication between two processes on the same host, the loopback of this data takes place below the IP layer, implying that all the TCP and IP processing takes place.

Figure 30.1 shows one possible scenario with three clients using one display. One client is on the same host as the server, using the Unix domain protocols. The other two clients are on different hosts, using TCP. One client is normally a *window manager* that has authority for the layout of windows on the display. The window manager allows us to move windows around the screen, or change their size, for example.

On first glance the terms *client* and *server* appear backward. With applications such as Telnet and FTP we think of the client as the interactive user at the keyboard and display. But with X, the keyboard and display belong to the server. Think of the server as the end providing the service. The service provided by X is access to a window, keyboard, and mouse. With Telnet the service is logging in to the remote host. With FTP the service is the filesystem on the server.

The X server is normally started when the X terminal or workstation is bootstrapped. The server creates a TCP end point and does a passive open on port $6000 + n$,

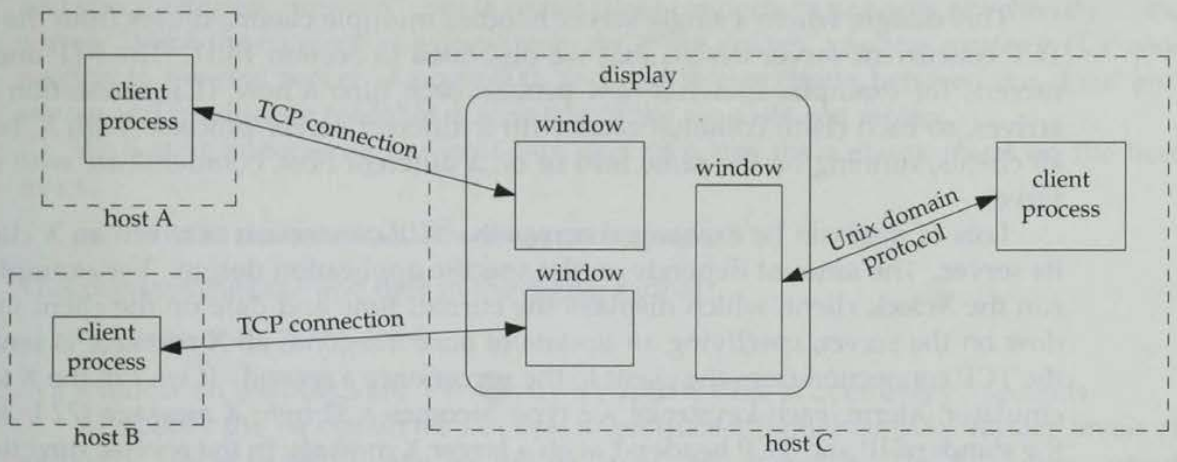


Figure 30.1 Three X clients using one display.

where n is the display number (normally 0). Most Unix servers also create a Unix domain socket with the name `/tmp/.X11-unix/Xn`, where n is again the display number.

When a client is started on another host, it creates a TCP end point and performs an active open to port $6000 + n$ on the server. Each client gets its own TCP connection to the server. It is the server's responsibility to multiplex all the clients. From this point on the client sends requests to the server across the TCP connection (e.g., create a window), the server sends back replies, and the server also sends events to the client (mouse button pushed, keyboard key pressed, window exposed, window resized, etc.).

Figure 30.2 is a redo of Figure 30.1, emphasizing that the clients communicate with the X server process, which in turn manages the windows on the display. Not shown here is that the X server also manages the keyboard and mouse.

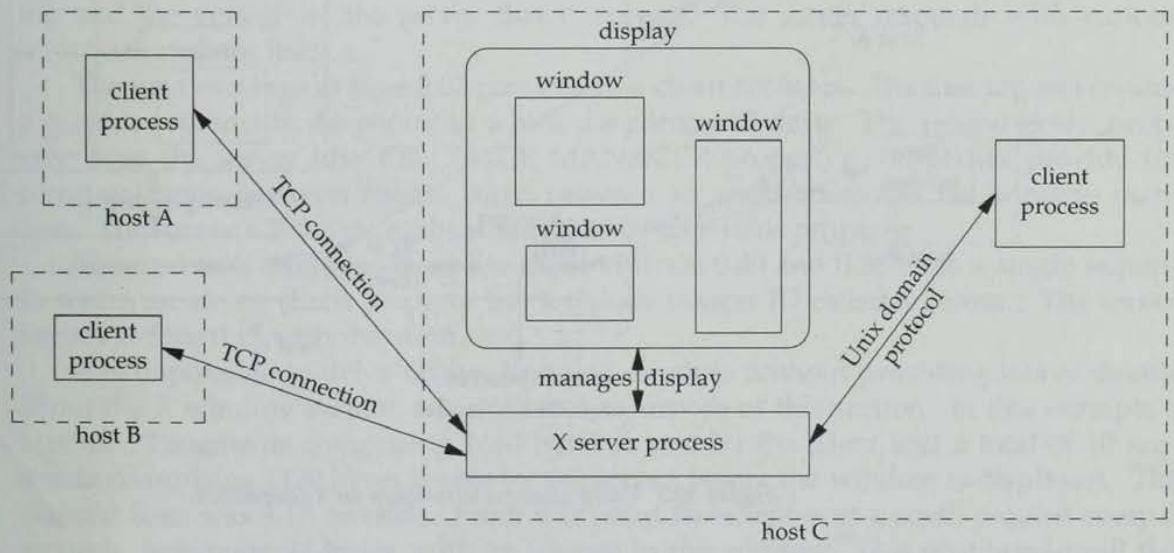


Figure 30.2 Three clients using one display.

This design, where a single server handles multiple clients, differs from the normal TCP concurrent server design that we described in Section 18.11. The FTP and Telnet servers, for example, spawn a new process each time a new TCP connection request arrives, so each client communicates with a different server process. With X, however, all clients, running on the same host or on a different host, communicate with a single server.

Lots of data can be exchanged across the TCP connection between an X client and its server. The amount depends on the specific application design. For example, if we run the Xclock client, which displays the current time and date on the client in a window on the server, specifying an update of once a second, an X message is sent across the TCP connection from the client to the server once a second. If we run the X terminal emulator, Xterm, each keystroke we type becomes a 32-byte X message (72 bytes with the standard IP and TCP headers), with a larger X message in the reverse direction with the character echo. [Droms and Dyksen 1990] measure the TCP traffic between various X clients and one particular server.

Xscope Program

A handy program for examining what's exchanged between an X client and its server is Xscope. It's provided with most X window implementations. It sits between a client and server, passing everything in both directions, and also deciphering all the client requests and server replies. Figure 30.3 shows the setup.

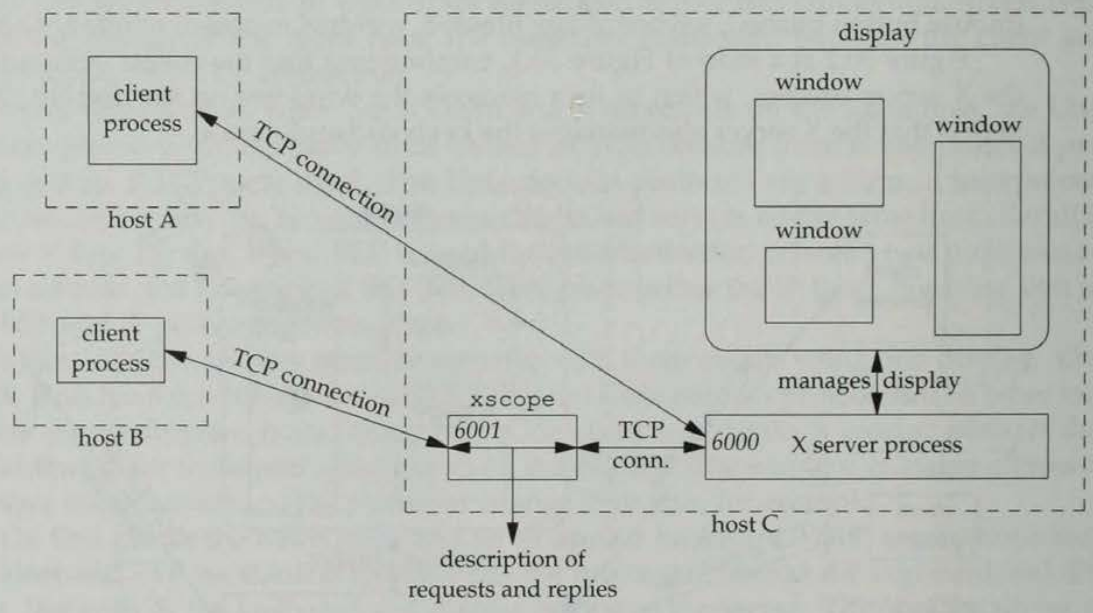


Figure 30.3 Using `xscope` to monitor an X connection.

We first start the `xscope` process on the same host as the server, but `xscope` listens for TCP connection requests on port 6001, not 6000. We then start a client on another host

and specify display number 1, not 0, so the client connects to `xscope`, not directly to the server. When the connection request from the client arrives, `xscope` creates a TCP connection to the real server at port 6000, and copies everything between the client and server, and produces a readable description of the requests and replies.

We'll start `xscope` on our host `sun` and then run the `xclock` client on the host `svr4`.

```
svr4 % DISPLAY=sun:1 xclock -digital -update 5
```

This displays the time and date in the digital format

```
Thu Sep  9 10:32:55 1993
```

in a window on the host `sun`. We specify an update time of once every 5 seconds.

We specify the `-q` option for `xscope` to produce minimal output. Various levels of verbosity are available, to see all the fields in each message. The following output shows the first three requests and replies.

```
sun % xscope -q
0.00: Client --> 12 bytes
0.02:                                  152 bytes <-- X11 Server
0.03: Client --> 48 bytes
        ............REQUEST: CreateGC
        ............REQUEST: GetProperty
0.20:                                  396 bytes <-- X11 Server
                                       ..............REPLY: GetProperty
0.30: Client -->  8 bytes
0.38: Client --> 20 bytes
        ............REQUEST: InternAtom
0.43:                                   32 bytes <-- X11 Server
                                       ..............REPLY: InternAtom
```

The first client message at time 0.00 and the server's response at time 0.02 are the standard connection setup between the client and server. The client identifies its byte ordering and the version of the server that it expects. The server responds with various information about itself.

The next message at time 0.03 contains two client requests. The first request creates a graphics context in the server in which the client will draw. The second gets a property from the server (the RESOURCE_MANAGER property). Properties provide for communication between clients, often between an application and the window manager. The server's 396-byte reply at time 0.20 contains this property.

The next two messages from the client at times 0.30 and 0.38 form a single request to return an atom. (Each property has a unique integer ID called an atom.) The server replies at time 0.43 with the atom.

It is impossible to delve farther into this example without providing lots of details about the X window system, which isn't the purpose of this section. In this example a total of 12 segments comprising 1668 bytes is sent by the client and a total of 10 segments comprising 1120 bytes is sent by the server, before the window is displayed. The elapsed time was 3.17 seconds. From this point the client sent a small request every 5 seconds, averaging 44 bytes, with an update to the window. This continued until the client was terminated.

LBX: Low Bandwidth X

The encoding used by the X protocol is optimized for LANs, where the time spent encoding and decoding the data is more important than minimizing the amount of data transmitted. While this is OK for an Ethernet, it becomes a problem for slow serial lines, such as SLIP and PPP links (Sections 2.4 and 2.6).

Work is progressing to define a standard called *Low Bandwidth X* (LBX) that uses the following techniques to reduce the amount of network traffic: caching, sending differences from previous packets, and compression. Specifications should appear early in 1994 with a sample implementation in the X window system Release 6.

30.6 Summary

The first two applications that we covered, Finger and Whois, are for obtaining information on users. Finger clients query a server, often to find someone's login name (for sending them mail) or to see if someone is currently logged in. The Whois client normally contacts the server run by the InterNIC, looking for information on a person, institution, domain, or network number.

The other Internet resource discovery services that we briefly described, Archie, WAIS, Gopher, Veronica, and WWW, help us locate files and documents across the Internet. Other resource discovery tools are currently being developed.

This chapter finished with a brief look at the X Window System, another heavy user of TCP/IP. We saw that the X server manages multiple windows on a display, and handles the communication between a client and its window. Each client has its own TCP connection to the server and a single server manages all the clients for a given display. With the Xscope program we saw how it's possible to place another program between a client and server to output information about the messages exchanged between the two.

Exercises

30.1 Use Whois to find the owner of the class A network ID 88.

30.2 Use Whois to find the DNS servers for the `whitehouse.gov` domain. Does the reply match the answer given by the DNS?

30.3 In Figure 30.3, do you think the `xscope` process must be run on the same host as the X server?

Appendix A

The tcpdump Program

The tcpdump program was written by Van Jacobson, Craig Leres, and Steven McCanne, all of Lawrence Berkeley Laboratory, University of California, Berkeley. Version 2.2.1 (June 1992) is used in this text.

tcpdump operates by putting the network interface card into *promiscuous mode* so that every packet going across the wire is captured. Normally interface cards for media such as Ethernet only capture link level frames addressed to the particular interface or to the broadcast address (Section 2.2).

The underlying operating system must allow an interface to be put into promiscuous mode and let a user process capture the frames. tcpdump support is provided or can be added to the following Unix systems: 4.4BSD, BSD/386, SunOS, Ultrix, and HP-UX. Consult the README file that accompanies the tcpdump distribution for the details on what operating system and which versions are supported.

There are alternatives to tcpdump. In Figure 10.8 (p. 135) we use the Solaris 2.2 program snoop to look at some packets. AIX 3.2.2 provides the program iptrace, which provides similar features.

A.1 BSD Packet Filter

Current BSD-derived kernels provide the *BSD Packet Filter* (BPF), which is one method used by tcpdump to capture and filter packets from a network interface that has been placed into promiscuous mode. BPF also works with point-to-point links, such as SLIP (Section 2.4), which require nothing special to capture all packets going through the interface, and with the loopback interface (Section 2.7).

BPF has a long history. The Enet packet filter was created in 1980 by Mike Accetta and Rick Rashid at Carnegie Mellon University. Jeffrey Mogul at Stanford ported the code to BSD and continued its development from 1983 on. Since then, it has evolved into the Ultrix Packet Filter at DEC, a STREAMS NIT module under SunOS 4.1, and BPF. Steven McCanne, of Lawrence Berkeley Laboratory, implemented BPF in Summer 1990. Much of the design is from Van Jacobson. Details of the latest version, and a comparison with Sun's NIT, are given in [McCanne and Jacobson 1993].

Figure A.1 shows the features of BPF when used with an Ethernet.

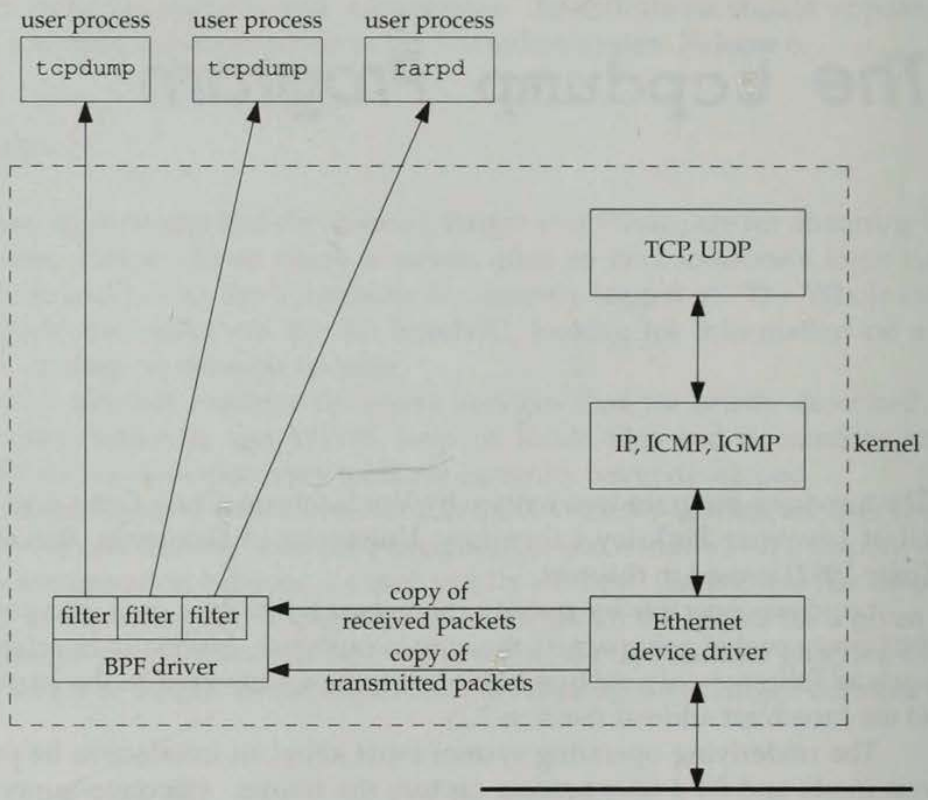


Figure A.1 BSD Packet Filter.

BPF places the Ethernet device driver into promiscuous mode and then receives a copy from the driver of each received packet and each transmitted packet. These packets are run through a user-specified filter, so that only packets that the user process considers interesting are passed to the process.

Multiple processes can be monitoring a given interface, and each process specifies its own filter. Figure A.1 shows two instances of `tcpdump` and an RARP daemon (Section 5.4) both monitoring the same Ethernet. Each instance of `tcpdump` specifies its own filter. The filter for `tcpdump` can be specified by the user on the command line, while `rarpd` always uses the same filter to capture only RARP requests.

In addition to specifying a filter, each user of BPF also specifies a timeout value. Since the data rate of the network can easily outrun the processing power of the CPU, and since it's costly for a user process to issue small reads from the kernel, BPF tries to

pack multiple frames into a single read buffer and return only when the buffer is full, or the user-specified timeout has expired. `tcpdump` sets the timeout to 1 second since it normally receives lots of data from BPF, while the RARP daemon receives few frames, so `rarpd` sets the timeout to 0 (which returns when a frame is received).

The user-specified filter to tell BPF what frames the process considers interesting is a list of instructions for a hypothetical machine. These instructions are interpreted by the BPF filter in the kernel. Filtering in the kernel, and not in the user process, reduces the amount of data that must pass from the kernel to the user process. The RARP daemon always uses the same filter program, which is built into the program. `tcpdump`, on the other hand, lets the user specify a filter expression on the command line each time it's run. `tcpdump` converts the user-specified expression into the corresponding sequence of instructions for BPF. Examples of the `tcpdump` expressions are:

```
% tcpdump tcp port 25
% tcpdump 'icmp[0] != 8 and icmp[0] != 0'
```

The first prints only TCP segments with a source or destination port of 25. The second prints only ICMP messages that are not echo requests or echo replies (i.e., not ping packets). This expression specifies that the first byte of the ICMP message, the *type* field from Figure 6.2, not equal 8 or 0, an echo request or echo reply from Figure 6.3. As you can see, fancy filtering requires knowledge of the underlying packet structure. The expression in the second example has been placed in single quotes to prevent the Unix shell from interpreting the special characters.

Refer to the `tcpdump`(1) manual page for complete details of the expression that the user can specify. The `bpf`(4) manual page details the hypothetical machine instructions used by BPF. [McCanne and Jacobson 1993] compare the design and performance of this machine against other approaches.

A.2 SunOS Network Interface Tap

SunOS 4.1.x provides a STREAMS pseudo-device driver called the *Network Interface Tap* or NIT. ([Rago 1993] contains additional details on streams device drivers. We'll call the feature "streams.") NIT is similar to the BSD Packet Filter, but not as powerful or as efficient. Figure A.2 shows the streams modules involved in using NIT. One difference between this figure and Figure A.1 is that BPF can capture packets received from and transmitted through the network interface, while NIT only captures packets received from the interface. Using `tcpdump` with NIT means we only see packets sent by other hosts on the network—we never see packets transmitted by our own host. (Although BPF works with SunOS 4.1.x, it requires source code changes to the Ethernet device driver, which are impossible for most users who don't have access to the source code.)

When the device `/dev/nit` is opened, the streams driver `nit_if` is opened. Since NIT is built using streams, processing modules can be pushed on top of the `nit_if` driver. `tcpdump` pushes the module `nit_buf` onto the STREAM. This module aggregates multiple network frames into a single read buffer, with the user process specifying a timeout value. This is similar to what we described with BPF. The RARP daemon doesn't push this module onto its stream, since it deals with a low volume of packets.

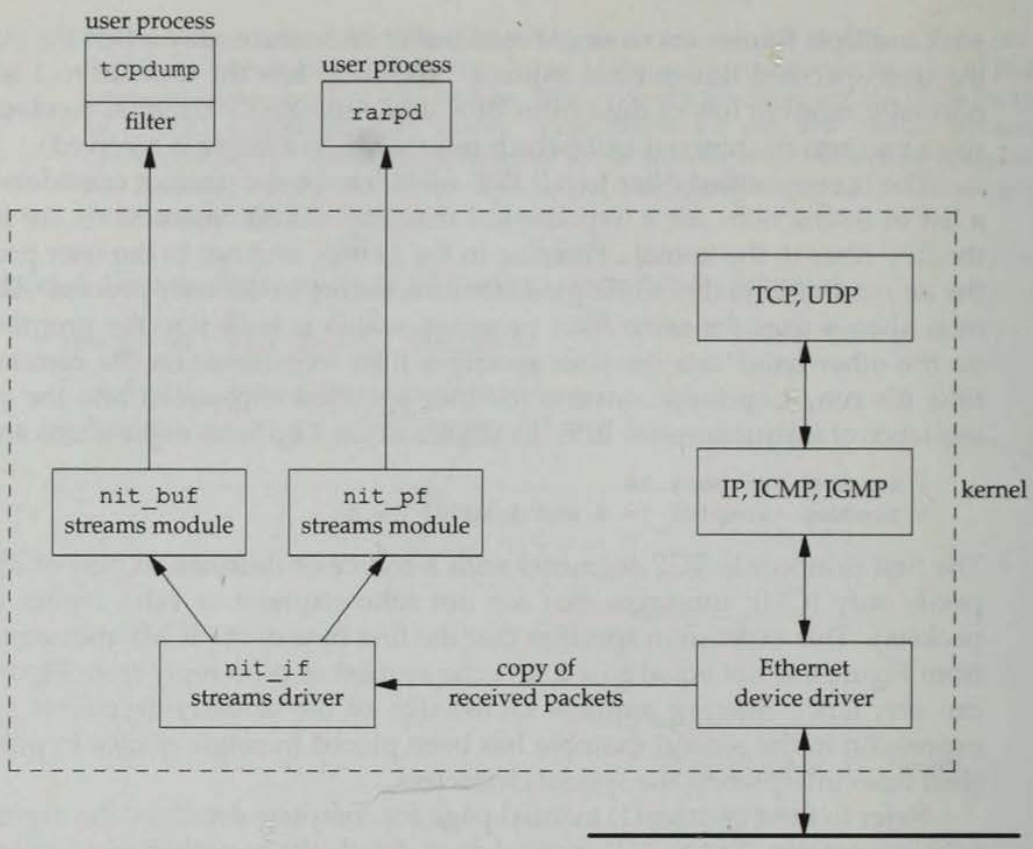


Figure A.2 SunOS Network Interface Tap.

The user-specified filtering is done by the streams module `nit_pf`. Notice in Figure A.2 that this module is used by the RARP daemon, but not by `tcpdump`. Instead, under SunOS `tcpdump` performs its own filtering in the user process. The reason is that the hypothetical machine instructions used by `nit_pf` are different (and not as powerful) as those supported by BPF. This means that when the user specifies a filter expression to `tcpdump` more data crosses the kernel-to-user boundary with NIT than with BPF.

A.3 SVR4 Data Link Provider Interface

SVR4 supports the *Data Link Provider Interface* (DLPI) which is a streams implementation of the OSI Data Link Service Definition. Most versions of SVR4 still support version 1 of the DLPI, SVR4.2 supports both versions 1 and 2, and Sun's Solaris 2.x supports version 2, with additional enhancements.

Network monitoring programs such as `tcpdump` must use the DLPI for raw access to the data-link device drivers. In Solaris 2.x the packet filter streams module has been renamed `pfmod` and the buffer module has been renamed `bufmod`.

Although Solaris 2.x is still new, an implementation of tcpdump should appear someday. Sun also supplies a program named snoop that performs functions similar to tcpdump. (snoop replaces the SunOS 4.x program named etherfind.) The author is not aware of any port of tcpdump to vanilla SVR4.

A.4 tcpdump Output

The output produced by tcpdump is "raw." We'll modify it for inclusion in the text to make it easier to read.

First, it always outputs the name of the network interface on which it is listening. We'll delete this line.

Next, the timestamp output by tcpdump is of the form 09:11:22.642008 on a system with microsecond resolution, or 09:11:22.64 on a system with only 10-ms clock resolution. (In Appendix B we talk more about computer clock resolution.) In either case the HH:MM:SS format is not what we want. Instead we are interested in both the relative time of each packet from the start of the dump, and the time difference between successive packets. We'll modify the output to show these two differences. The first difference we print with six digits to the right of the decimal point when microsecond resolution is available (two digits when only 10-ms resolution is provided), and the second difference we print with either four digits or two digits to the right of the decimal point (depending on the clock resolution).

> In this text most tcpdump output was collected on the host sun, which provides microsecond resolution. Some output was collected on the host bsdi running BSD/386 Version 0.9.4, which only provided 10-ms resolution (e.g., Figure 5.1). Some output was also collected on bsdi when it was running BSD/386 Version 1.0, which provides microsecond resolution.

tcpdump always prints the name of the sending host, then a greater than sign, then the name of the destination host. This makes it hard to follow the flow of packets between two hosts. Although our tcpdump output will still show the direction of data flow like this, we'll often take this output and produce a time line instead. (The first of these in the text is Figure 6.11, p. 80.) In our time lines one host will be on the left, and the other on the right. This makes it easier to see which side sends and which side receives each packet.

We add line numbers to the tcpdump output, allowing us to reference specific lines in the text. We also add additional space between certain lines, to separate some packet exchanges.

Finally, tcpdump output can exceed the width of the page. We wrap long lines around at convenient points in the line.

As an example, the output produced by tcpdump corresponding to Figure 4.4 (p. 58) is shown in Figure A.3, assuming an 80-column terminal window.

We won't show our typing the interrupt key (which terminates tcpdump) and we won't show the number of packets received and dropped. (Dropped packets are those that arrived faster than tcpdump could keep up with. Since the examples in the text were often run on an otherwise idle network, this is always 0.)

```
sun % tcpdump -e
tcpdump: listening on le0
09:11:22.642008 0:0:c0:6f:2d:40 ff:ff:f f:ff:ff:ff arp 60: arp who-has svr4 tell
bsdi
09:11:22 .644182 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 arp 60: arp reply svr4 is-at 0:0
:c0:c2:9b:26
09:11:22.644839 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26  ip 60: bsdi.1030 > svr4.discard:
 S 596459521:596459521(0) win 4096 <mss 1024> [tos 0x10]
09:11:22. 649842 0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 ip 60: svr4.discard > bsdi.1030:
 S 3562228225:3562228225(0) ack 596459522 win 4096 <mss 1024>
09:1 1:22.651623 0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60: bsdi.1030 > svr4.discard:
 . ack 1 win 4096 [tos 0x10]
                                        4 other packets that we don't show
^?                                      type our interrupt key to terminate
9 packets received by filter
0 packets dropped by kernel
```

Figure A.3 tcpdump output for Figure 4.4 (p. 58).

A.5 Security Considerations

It should be obvious that tapping into a network's traffic lets you see many things you shouldn't see. For example, the passwords typed by users of applications such as Telnet and FTP are transmitted across the network exactly as the user enters them. (This is called the *cleartext* representation of the password, in comparison to the *encrypted* representation. It is the encrypted representation that is stored in the Unix password file, normally /etc/passwd or /etc/shadow.) Nevertheless, there are many times when a network administrator needs to use a tool such as tcpdump to diagnose network problems.

Our use of tcpdump is as a learning tool, to see what really gets transmitted across the network. Access to tcpdump, and similar vendor-supplied utilities, depends on the system. Under SunOS, for example, access to the NIT device is restricted to the superuser. The BSD Packet Filter uses a different technique: access is controlled by the permissions on the devices /dev/bpfXX. Normally these devices are readable and writable only by the owner (which should be the superuser) and readable by the group (often the system administration group). This means normal users can't run programs such as tcpdump, unless the system administrator makes the program set-user-ID.

A.6 Socket Debug Option

Another way to see what's going on with a TCP conne ction is to enable socket debugging, on systems that support this feature. This feature works only with TCP (not with other protocols) and requires application support (to enable a socket option when it's started).

Most Berkeley-derived implementations support thi s, including SunOS, 4.4BSD, and SVR4.

The program enables a socket option, and the kernel then keeps a trace record of what happens on that connection. At some later time all this information can be output by running the program `trpt`(8). It doesn't require special permission to enable the socket debug option, but it requires special privileges to run `trpt`, since it accesses the kernel's memory.

Our `sock` program (Appendix C) supports this feature with its `-D` option, but the information output is harder to decipher and understand than the corresponding `tcpdump` output. We do, however, use it in Section 21.4 to look at kernel variables in the TCP connection block that `tcpdump` cannot access.

Appendix B

Computer Clocks

Since most of the examples in this text measure a time interval, we need to describe in more detail the type of timekeeping used by current Unix systems. The following description applies to the systems being used for the examples in this book, and for most Unix systems. Additional details are given in Sections 3.4 and 3.5 of [Leffler et al. 1989].

The hardware generates a clock interrupt at some frequency. For Sun SPARCs and Intel 80386s the interrupts occur every 10 ms.

It should be noted that most computers use an uncompensated crystal oscillator to generate these interrupts. As noted in Table 7 of RFC 1305 [Mills 1992], you don't want to ask what the drift per day of such an oscillator is. This means few computers keep accurate time (i.e., the interrupts don't occur exactly every 10 ms). A 0.01% tolerance gives an error of 8.64 seconds per day. To keep better time requires (1) a better oscillator, (2) an external time source with greater precision (e.g., the time source supplied by the Global Positioning Satellites), or (3) access across the Internet to systems with more precise clocks. The latter is provided by the Network Time Protocol, as described in detail in RFC 1305, which is beyond the scope of this book.

Another common source of time errors in Unix systems is that the 10-ms clock interrupts only cause the kernel to increment a variable that keeps track of the time. If the kernel loses an interrupt (i.e., it's too busy for the 10 ms between two consecutive interrupts), the clock will lose 10 ms. Lost interrupts of this type often cause Unix systems to lose time.

Even though the clock interrupts arrive approximately every 10 ms, newer systems such as SPARCs provide a higher resolution timer to measure time differences. `tcpdump`, through the NIT driver (described in Appendix A) has access to this higher resolution timer. On SPARCs this timer provides microsecond resolution. Access to this higher resolution timer is also provided for user processes through the `gettimeofday`(2) function.

The author ran the following experiment. A program was run that called the `gettimeofday` function 10,000 times in a loop, saving each return value in an array. At the end of the loop the 9,999 differences were printed out. For a SPARC ELC the distribution of the differences are shown in Figure B.1.

Microseconds	Count
36	4,914
37	4,831
38	167
39	8
other	79

Figure B.1 Distribution of time required to call `gettimeofday` 10,000 times on SPARC ELC.

The total clock time required to run the program was 0.38 seconds, on an otherwise idle system. From this we can say that the time for the process to call `gettimeofday` is about 37 microseconds. Since the ELC is rated around 21 MIPS (million instructions per second), 37 microseconds corresponds to about 800 instructions. This seems reasonable for the kernel to handle a system call from a user process, execute the system call, copy back 8 bytes of results, and return to the user process. (MIPS ratings are questionable, and it's hard to try to measure instruction times on current systems. All we're trying to do is get a rough idea and see if the values make sense.)

From this simple experiment we can say that the values returned by `gettimeofday` do contain microsecond resolution.

If we run similar tests under SVR4/386, however, the results are different. This is because many 386 Unix systems, such as SVR4, only count the 10-ms clock interrupts, and don't try to provide any higher resolution. Figure B.2 is the distribution of the 9,999 differences under SVR4 on an 25 Mhz 80386.

Microseconds	Count
0	9,871
10,000	128

Figure B.2 Distribution of time required to call `gettimeofday` 10,000 times under SVR4/386.

These values are worthless, since the differences are normally less than 10 ms, which is treated as 0. About all we can do on these systems is measure the clock time on an idle system, and divide by the number of loops. This provides an upper bound, since it includes the time required to call `printf` 9,999 times, writing the results to a file. (In the SPARC case, Figure B.1, the differences did not include the `printf` times since all 10,000 values were first obtained, and then the results were printed.) Under SVR4 the clock time was 3.15 seconds, yielding 315 microseconds per system call. This system call time, about 8.5 times slower than the SPARC, seems about right.

BSD/386 Version 1.0 provides microsecond resolution similar to the SPARC. It reads the 8253 clock register and calculates the number of microseconds since the last clock tick. This is made available to processes that call `gettimeofday` and to kernel modules such as the BSD Packet Filter.

In relation to `tcpdump` these numbers mean that we can believe the millisecond and submillisecond values that are printed on the SPARC and BSD/386 systems, but the values printed by `tcpdump` under SVR4/386 will always be a multiple of 10 ms. For other programs that print round-trip times, such as `ping` (Chapter 7) and `traceroute` (Chapter 8), on the SPARC and BSD/386 systems we can believe the millisecond values that are output, but the values printed under SVR4/386 will always be multiples of 10. To measure anything like the ping time on a LAN, which we show in Chapter 7 to be around 3 ms, requires running `ping` on the SPARC or BSD/386.

Some of the examples in this text were run under BSD/386 Version 0.9.4, which was similar to SVR4 in that it provided only 10-ms clock resolution. When we show `tcpdump` output from this system, we show only two numbers to the right of the decimal point, since that's the resolution provided.

Appendix C

The `sock` Program

A simple test program named `sock` is used throughout the book to generate TCP and UDP data. It is used as both a client and server process. Having a test program like this, which is executable from a shell prompt, prevents us from having to write new client and server C programs for each specific feature that we want to examine. Since the purpose of this book is to understand the networking protocols, and not network programming, in this Appendix we only describe the program and its various options.

> There are numerous other programs with functionality similar to `sock`. Juergen Nickelsen wrote a program named `socket` and Dave Yost wrote a program named `sockio`. Both contain many similar features. Pieces of the `sock` program have also been inspired by the public domain `ttcp` program, written by Mike Muuss and Terry Slattery.

The `sock` program operates in one of four modes:

1. Interactive client: the default. The program connects to a server and then copies standard input to the server and copies everything received from the server to standard output. This is shown in Figure C.1.

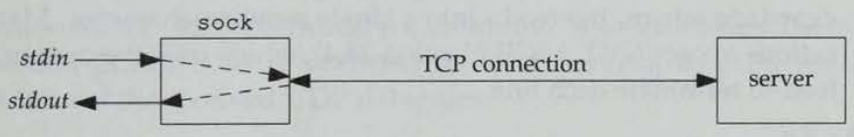


Figure C.1 Default operation of `sock` as interactive client.

We must specify the name of the server host and the name of the service to connect to. The host can also be specified as a dotted-decimal number, and the service can be specified as an integer port number. Connecting to the standard echo server (Section 1.12), from `sun` to `bsdi` echoes everything we type:

```
sun % sock bsdi echo
a test line                    we type this line
a test line                    and the echo server returns a copy
^D                             type our end-of-file character to terminate
```

2. Interactive server: the `-s` option is specified. The service name (or port number) is required:

```
sun % sock -s 5555             act as server listening on port 5555
```

The program waits for a connection from a client and then copies standard input to the client and copies everything received from the client to standard output. An Internet address can precede the port number on the command line, to specify on which local interface connections are accepted:

```
sun % sock -s 140.252.13.33 5555      accept connections only on Ethernet
```

The default mode is to accept a connection request on any local interface.

3. Source client: the `-i` option is specified. By default a 1024-byte buffer is written to the network 1024 times. The `-n` and `-w` options can change these defaults. For example,

```
sun % sock -i -n12 -w4096 bsdi discard
```

writes 12 buffers, each containing 4096 bytes of data, to the discard server on host `bsdi`.

4. Sink server: the `-i` and `-s` options are specified. Data is read from the network and discarded.

Although these examples used TCP (the default), the `-u` option specifies UDP.

There are a multitude of options that provide finer control over exactly how the program operates. These options are needed to generate all the test conditions used throughout the text.

`-b` *n* Bind *n* as the client's local port number. (By default an ephemeral port number assigned by the system is used by the client.)

`-c` Convert newline characters that are read on standard input into a carriage return and a linefeed. Similarly, when reading from the network, convert the sequence <carriage return, linefeed> into a single newline character. Many Internet applications expect NVT ASCII (Section 26.4), which uses the carriage return and linefeed to terminate each line.

`-f` *a.b.c.d.p*
Specify the foreign IP address (*a.b.c.d*) and the foreign port number (*p*) for a UDP end point (Section 11.12).

`-h` Implement TCP's half-close facility (Section 18.5). That is, do not terminate when an end-of-file is encountered on standard input. Instead, issue a half-close on the TCP connection but continue reading from the network until the peer closes the connection.

`-i` Source client or sink server. Either write data to the network (default) or if used in conjunction with the `-s` option, read data from the network. The `-n` option can specify the number of buffers to write (or read), the `-w` option can specify the size of each write, and the `-r` option can specify the size of each read.

`-n` *n* When used with the `-i` option, *n* specifies the number of buffers to read or write. The default value of *n* is 1024.

`-p` *n* Specify the number of seconds to pause between each read or write. This can be used with the source client (`-i`) or sink server (`-is`) to delay between each read or write of network. Also see the `-P` option to pause before the first read or write.

`-q` *n* Specify the size of the pending connection queue for the TCP server: the number of accepted connections that TCP will queue for the application (Figure 18.23). The default is 5.

`-r` *n* When used with the `-is` options, *n* specifies the size of each read from the network. The default is 1024 bytes per read.

`-s` Operate as a server instead of as a client.

`-u` Use UDP instead of TCP.

`-v` Verbose. Print additional details (such as the client and server ephemeral port numbers) onto standard error.

`-w` *n* When used with the `-i` option, specifies the size of each write to the network. The default is 1024 bytes per write.

`-A` Enable the `SO_REUSEADDR` socket option. With TCP this allows the process to assign itself a port number that is part of a connection that is in the 2MSL wait. With UDP on a system that supports multicasting, it allows multiple processes to use the same local port to receive broadcast or multicast datagrams.

`-B` Enable the `SO_BROADCAST` socket option to allow UDP datagrams to be sent to a broadcast IP address.

`-D` Enable the `SO_DEBUG` socket option. This causes additional debugging information to be maintained by the kernel for this TCP connection (Section A.6). This information can be output later by running the `trpt`(8) program.

`-E` Enable the `IP_RECVDSTADDR` socket option, if supported by the implementation (Section 11.12). This is intended for UDP servers, to print the destination IP address of the received UDP datagram.

`-F` Specifies a concurrent TCP server. That is, the server creates a new process using the `fork` function for each client connection.

`-K` Enable TCP's `SO_KEEPALIVE` socket option (Chapter 23).

`-L` *n* Set the linger time (`SO_LINGER` socket option) for a TCP end point to *n*. A linger time of 0 means when the network connection is closed, any data still queued for sending is discarded and a reset is sent to the peer (Section 18.7).

A positive linger time is the time (in 100ths of a second) that a close on the network connection should wait for all outstanding data to be sent and acknowledged. If, after closing the network connection, all the pending data has not been sent and acknowledged when this timer expires, the close will return an error.

`-N` Set the `TCP_NODELAY` socket option to disable the Nagle algorithm (Section 19.4).

`-O` *n* Specify the number of seconds for a TCP server to pause before accepting the first client connection.

`-P` *n* Specify the number of seconds to pause before the first read or write of the network. This can be used with the sink server (`-is`) to delay after accepting the connection request from the client but before performing the first read from the network. When used with the sink source (`-i`) it delays after the connection has been established, but before the first write to the network. Also see the `-p` option to pause between each successive read or write.

`-Q` *n* Specify the number of seconds for a TCP client or server to pause after receiving an end-of-file from the other end, but before closing its end of the connection.

`-R` *n* Set the socket's receive buffer (`SO_RCVBUF` socket option) to *n*. This can directly affect the size of the receive window advertised by TCP. With UDP this specifies the largest UDP datagram that can be received.

`-S` *n* Set the socket's send buffer (`SO_SNDBUF` socket option) to *n*. With UDP this specifies the largest UDP datagram that can be sent.

`-U` *n* Enter TCP's urgent mode after write number *n* to the network. One byte of data is written to initiate urgent mode (Section 20.8).

Appendix D

Solutions to Selected Exercises

Chapter 1

1.1 The value is $2^7 - 2$ (126) plus $2^{14} - 2$ (16,382) plus $2^{21} - 2$ (2,097,150) for a total of 2,113,658. We subtract 2 in each calculation since a network ID of all zero bits or all one bits is invalid.

1.2 Figure D.1 shows a plot of the values through August 1993.

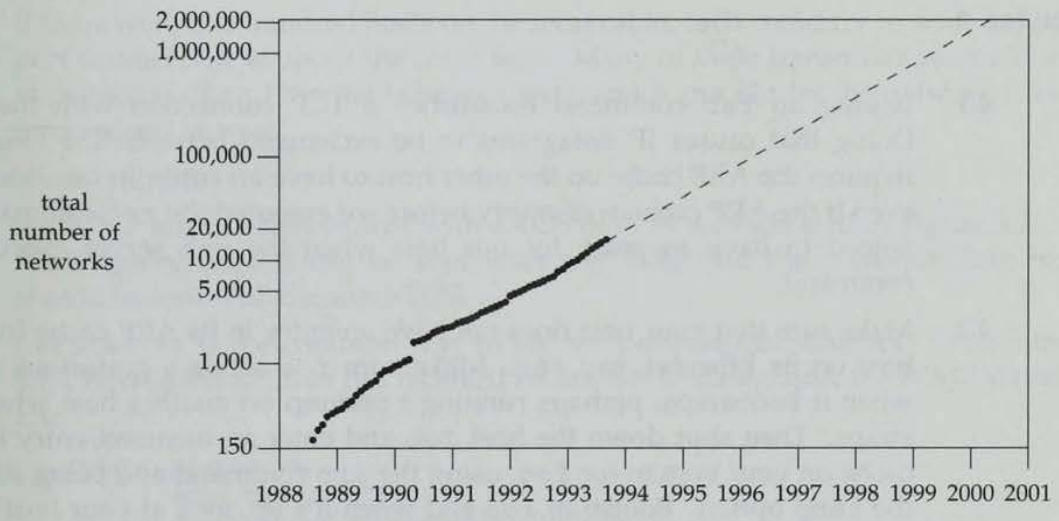


Figure D.1 Number of networks announced to NSFNET.

The dashed line estimates that the maximum number of networks will be reached in the year 2000, if the exponential growth continues.

1.3 "Be liberal in what you accept, and conservative in what you send."

Chapter 3

3.1 No, any class A address with a network ID of 127 is OK, although most systems use 127.0.0.1.

3.2 `kpno` has five interfaces: three point-to-point links and two Ethernets. R10 has four Ethernet interfaces. `gateway` has three interfaces: two point-to-point links and one Ethernet. Finally, `netb` has one Ethernet interface and two point-to-point links.

3.3 There's no difference: both have a subnet mask of 255.255.255.0, as does a class C address that is not subnetted.

3.5 It's valid and it's called a *noncontiguous subnet mask* since the 16 bits for the subnet mask are not contiguous. The RFCs, however, recommend against using noncontiguous subnet masks.

3.6 It's a historical artifact. The value is 1024 + 512 but the MTU values printed include any required headers. Solaris 2.2 sets the MTU of the loopback interface to 8232 (8192 + 40), which allows room for 8192 bytes of user data along with the normal 20-byte IP header and 20-byte TCP header.

3.7 First, datagrams eliminate the need for connection state in the routers. Second, datagrams provide the basic building block on which unreliable (UDP) and reliable (TCP) transport layers can be built. Third, datagrams represent the minimal network layer assumption, allowing a wide range of data-link layers to be used.

Chapter 4

4.1 Issuing an `rsh` command establishes a TCP connection with the other host. Doing that causes IP datagrams to be exchanged between the two hosts. This requires the ARP cache on the other host to have an entry for our host. Therefore, even if the ARP cache was empty before we executed the `rsh` command, it's guaranteed to have an entry for our host when the `rsh` server executes the `arp` command.

4.2 Make sure that your host does not have an entry in its ARP cache for some other host on its Ethernet, say `foo`. Make sure `foo` sends a gratuitous ARP request when it bootstraps, perhaps running `tcpdump` on another host when `foo` bootstraps. Then shut down the host `foo` and enter an incorrect entry into the ARP cache on your system for `foo`, using the `arp` command and being sure to specify the `temp` option. Bootstrap `foo` and when it's up, look at your host's ARP cache entry for it to see whether the incorrect entry has been corrected.

4.3 Read Section 2.3.2.2 of the Host Requirements RFC and Section 11.9 of this text.

4.4 Assuming that a completed ARP entry existed for the server on the client when the server was taken down, if we continually try to contact the (down) server, the

ARP timeout gets extended for another 20 minutes. When the server finally reboots with a new hardware address, if it doesn't issue a gratuitous ARP, the old, invalid ARP entry will still exist on the client. We won't be able to contact the server at its new hardware address until we either manually delete the ARP cache entry or stop trying to contact it for 20 minutes.

Chapter 5

5.1 A separate frame type is not an absolute requirement, since the *op* field in Figure 4.3 has a different value for all four operations (ARP request, ARP reply, RARP request, and RARP reply). But the implementation of an RARP server, separate from the kernel's ARP server, is made easier with the different frame type field.

5.2 Each RARP server can delay for a small random time before sending a response.

As a refinement, one RARP server can be designated the primary and the others as secondaries. The primary server can respond without a delay, and the secondaries with random delays.

As yet another refinement, with a primary and secondaries, the secondaries can be programmed to respond only to a duplicate request received in a short time frame. This assumes that the reason for the duplicate request is that the primary is down.

Chapter 6

6.1 If there were one hundred hosts on the local cable, each could try to send an ICMP port unreachable at about the same time. Many of these transmissions could lead to collisions (if an Ethernet is being used), which can render the network useless for a second or two.

6.2 It is a "should."

6.3 An ICMP error is always sent with a TOS of 0, as we indicated in Figure 3.2. An ICMP query request can be sent with any TOS, and the corresponding reply should be sent with the same TOS.

6.4 `netstat -s` is the common way to see the per-protocol statistics. On a SunOS 4.1.1 host (`gemini`) that has received 48 million IP datagrams, the ICMP statistics are:

```
Output histogram:
        echo reply: 1757
        destination unreachable: 700
        time stamp reply: 1
Input histogram:
        echo reply: 211
        destination unreachable: 3071
        source quench: 249
```

```
routing redirect: 2789
echo: 1757
#10: 21
time exceeded: 56
time stamp: 1
```

The 21 input messages of type 10 are router solicitations that SunOS 4.1.1 doesn't support.

SNMP can also be used (Figure 25.26) and some systems, such as Solaris 2.2, generate `netstat -s` output that uses SNMP variable names.

Chapter 7

7.2 86 bytes divided by 960 bytes/sec, times 2 gives 179.2 ms. When `ping` is run at this speed, the printed values are 180 ms.

7.3 (86 + 48) bytes divided by 960 bytes/sec, times 2 gives 279.2 ms. The additional 48 bytes are because the final 48 bytes of the 56 bytes in the data portion must be escaped: `0xc0` is the SLIP END character.

7.4 CSLIP only compresses the TCP and IP headers for TCP segments. It has no effect on the ICMP messages used by `ping`.

7.5 On a SPARCstation ELC a ping of the loopback address yields an RTT of 1.310 ms, while a ping of the host's Ethernet address yields an RTT of 1.460 ms. This difference is the additional processing done by the Ethernet driver, to determine that the datagram is really destined for the local host. You need a version of `ping` that outputs microsecond resolution to measure this.

Chapter 8

8.1 If an incoming datagram has a TTL of 0, doing the decrement and then test would set the TTL to 255 and let the datagram continue. Although a router should never receive a datagram with a TTL of 0, it has occurred.

8.2 We noted that `traceroute` stores 12 bytes of data in the data portion of the UDP datagram, part of which is the time the datagram was sent. From Figure 6.9, however, we see that ICMP only returns the first 8 bytes of the IP datagram that was in error, and we noted there that this is the 8-byte UDP header. Therefore the time value stored by `traceroute` is not returned in the ICMP error message. `traceroute` saves the time when it sends a packet, and when an ICMP reply is received, fetches the current time and subtracts the two value to get the RTT.

Recall from Chapter 7 that `ping` stored the time in the outgoing ICMP echo request and this data was echoed by the server. This allowed `ping` to print the correct RTT, even if the packets were returned out of order.

8.3 The first line of output is correct and identifies R1. The next probe starts with a TTL of 2, and this is decremented by R1. When R2 receives this it decrements the

TTL from 1 to 0 but incorrectly forwards it to R3. R3 sees that the incoming TTL is 0 and sends back the time exceeded. This means the second line of output (for the TTL of 2) identifies R3, not R2. The third line of output correctly identifies R3. The clue that this bug is present is two consecutive lines of output that identify the same router.

8.4 In this case the TTL of 1 identifies R1, the TTL of 2 identifies R2, and the TTL of 3 identifies R3; but when the TTL is 4 the UDP datagram gets to the destination with an incoming TTL of 1. The ICMP port unreachable is generated, but its TTL is 1 (incorrectly copied from the incoming TTL). This ICMP message goes to R3 where the TTL is decremented and the message discarded. An ICMP time exceeded is *not* generated, since the datagram that was discarded was an ICMP error message (port unreachable). A similar scenario occurs for the probe with a TTL of 5, but this time the outgoing port unreachable starts with a TTL of 2 (the incoming TTL) and makes it back to R2, where it's discarded. The port unreachable corresponding to the probe with a TTL of 6 makes it back to R1, where it's discarded. Finally the port unreachable for the probe with a TTL of 7 makes it all the way back, where it arrives with an incoming TTL of 1. (`traceroute` considers an arriving ICMP message with a TTL of 0 or 1 to be suspicious, so it prints an exclamation point after the RTT.) In summary, the lines for a TTL of 1, 2, and 3 correctly identify R1, R2, and R3, followed by three lines each containing three timeouts, followed by the line for a TTL of 7 that identifies the destination.

8.5 It appears that all these routers initialize the outgoing TTL of an ICMP message to 255. This is common. The incoming value of 255 from `netb` is what we expect, but the value of 253 from `butch` means there is probably a missing router between it and `netb`. Otherwise we would expect an incoming TTL of 254 at this point. Similarly, from `enss142.UT.westnet.net` we expect a value of 252, not 249. It appears these missing routers are not handling the outbound UDP datagram correctly, but they are decrementing the TTL on the returned ICMP message correctly.

We must be careful when looking at the incoming TTL, since sometimes a value other than what we expect can be caused by the return ICMP message taking a different path from the outbound UDP datagram. In this example, however, it confirms what we suspect—there are missing routers that `traceroute` is not finding when the loose source routing option is used.

8.7 The `ping` client sets the identifier field in the ICMP echo request message (Figure 7.1, p. 86) to its process ID. The ICMP echo reply contains this identifier field. Each client looks at this returned identifier field and handles only those that it sent.

The `traceroute` client sets its UDP source port number to the logical-OR of its process ID and 32768. Since the returned ICMP message always contains the first 8 bytes of the IP datagram that generated the error (Figure 6.9, p. 78), which includes the entire UDP header, this source port number is returned in the ICMP error.

8.8 The `ping` client sets the optional data portion of the ICMP echo request message to the time at which the packet is sent. This optional data must be returned in the ICMP echo reply. This allows `ping` to calculate the accurate round-trip time, even if packets are returned out of order.

The `traceroute` client can't operate this way because all that's returned in the ICMP error is the UDP header (Figure 6.9, p. 78), none of the UDP data. Therefore `traceroute` must remember when it sends a request, wait for the reply, and calculate the time difference.

This illustrates another difference between Ping and Traceroute: Ping sends one packet a second, regardless of whether it receives any replies, while Traceroute sends a request and then waits for either a reply or a timeout before sending the next request.

8.9 Since Solaris 2.2 starts ephemeral UDP port numbers at 32768 by default, there is a much greater chance that the destination port is in use on the destination host.

Chapter 9

9.1 When the ICMP standard was first specified, RFC 792 [Postel 1981b], subnetting was not in use. Also, using a single network redirect instead of N host redirects (for all N hosts on the destination network) saves some space in the routing table.

9.2 The entry is not required, but if it is removed, all IP datagrams to `slip` are sent to the default router (`sun`), which then forwards them to the router `bsdi`. Since `sun` is forwarding a datagram out the same interface on which it was received, it sends an ICMP redirect to `svr4`. This creates the same routing table entry on `svr4` that we removed, although this time it is created by a redirect instead of being added at bootstrap time.

9.3 When the 4.2BSD host receives the datagram destined for 140.1.255.255 it finds that it has a route to the network (140.1) so it tries to forward the datagram. To do this it sends an ARP broadcast looking for 140.1.255.255. No reply is received for this ARP request, so the datagram is eventually discarded. If there are many of these 4.2BSD hosts on the cable, every one sends out this ARP broadcast at about the same time, swamping the network temporarily.

9.4 This time a reply is received for each ARP request, telling each 4.2BSD host to send the datagram to the specified hardware address (the Ethernet broadcast). If there are k of these 4.2BSD hosts on the cable, all receive their own ARP reply, causing each one to generate another broadcast. Each host receives each broadcast IP datagram destined to 140.1.255.255, and since every host now has an ARP cache entry, the datagram is forwarded again to the broadcast address. This continues and generates an *Ethernet meltdown*. [Manber 1990] describes other forms of chain reactions in networks.

Chapter 10

10.1 Thirteen of the routes came from `kpno`: all except 140.252.101.0 and 140.252.104.0, the other networks to which `gateway` is directly connected.

10.2 Sixty seconds will pass before the 25 routes advertised in the lost datagram are updated. This isn't a problem because RIP normally requires 3 minutes without an update before it declares a route dead.

10.3 RIP runs on top of UDP, and UDP provides an optional checksum for the data portion of the UDP datagram (Section 11.3). OSPF, however, runs on top of IP. The IP checksum covers only the IP header, so OSPF must add its own checksum field.

10.4 Load balancing increases the chances of packets being delivered out of order, and possibly distorts the round-trip times calculated by the transport layer.

10.5 This is called simple split horizon.

10.6 In Figure 12.1 we show that each of the 100 hosts processes the broadcast UDP datagram through the device driver, IP layer, and UDP layer, where it'll finally be discarded when it's discovered that UDP port 520 is not in use.

Chapter 11

11.1 Since there are 8 additional bytes of header when IEEE 802 encapsulation is used, 1465 bytes of user data is the smallest size that causes fragmentation.

11.3 There are 8200 bytes of data for IP to send, the 8192 bytes of user data and the 8-byte UDP header. Using the `tcpdump` notation, the first fragment is 1480@0+ (1480 bytes of data, offset of 0, with the "more fragments" bit set). The second is 1480@1480+, the third is 1480@2960+, the fourth is 1480@4440+, the fifth is 1480@5920+, and the sixth is 800@7400. $1480 \times 5 + 800 = 8200$, which is the number of bytes to send.

11.4 Each 1480-byte fragment is divided into three pieces: two 528-byte fragments and one 424-byte fragment. The largest multiple of 8 less than 532 $(552 - 20)$ is 528. The 800-byte fragment is divided into two pieces: a 528-byte fragment and a 272-byte fragment. Thus, the original 8192-byte datagram becomes 17 frames across the SLIP link.

11.5 No. The problem is that when the application times out and retransmits, the IP datagram generated by the retransmission has a new identification field. Reassembly is done only for fragments with the same identification field.

11.6 The identification field in the IP header (47942) is the same.

11.7 First, from Figure 11.4 we see that `gemini` does not have outgoing UDP checksums enabled. It's highly probable that the operating system on this host (SunOS 4.1.1) is one that never verifies an incoming UDP checksum unless outgoing UDP

checksums are enabled. Second, it could be that most of the UDP traffic is local traffic, instead of WAN traffic, and therefore not subject to all the vagaries of WANs.

11.8 The loose and strict source routing options are copied into each fragment. The timestamp option and the record route option are not copied into each fragment—they appear only in the first fragment.

11.9 No. We saw in Section 11.12 that many implementations can filter incoming datagrams destined for a given UDP port number based on the destination IP address, source IP address, and source port number.

Chapter 12

12.1 Broadcasting by itself does not add to network traffic, but it adds extra host processing. Broadcasting can lead to additional network traffic if the receiving hosts incorrectly respond with errors such as ICMP port unreachables. Also, routers normally don't forward broadcast packets, whereas bridges normally do, so broadcasts on a bridged network can travel much farther than they would on a routed network.

12.2 Every host receives a copy of every broadcast. The interface layer receives the frame, and passes it to the device driver. If the type field is for some other protocol, it is the device driver that discards the frame.

12.3 First execute `netstat -r` to see the routing table. This shows the names of all the interfaces. Then execute `ifconfig` (Section 3.8) for each interface: the flags tell if the interface supports broadcasting, and if so the broadcast address is also output.

12.4 Berkeley-derived implementations do not allow a broadcast datagram to be fragmented. When we specified the length of 1472 bytes, the resulting IP datagram was exactly 1500 bytes, the Ethernet MTU. Refusing to allow a broadcast datagram to be fragmented is a policy decision—there is no technical reason (other than a desire to reduce the number of broadcast packets).

12.5 Depending on the multicasting support in the various Ethernet interface cards in the 100 hosts, the multicast datagram can be ignored by the interface card, or discarded by the device driver.

Chapter 13

13.1 Use some host-unique value when generating the random value. The IP address and link-layer address are two values that should differ on every host. The time-of-day is a bad choice, especially if all the hosts run a protocol such as NTP to synchronize their clocks.

13.2 They added an application protocol header that included a sequence number and a timestamp.

Chapter 14

14.1 A resolver is always a client, but a name server is both a client and server.

14.2 The question is returned, which accounts for the first 44 bytes. The single answer occupies the remaining 31 bytes: a 2-byte pointer for the domain name (i.e., a pointer to the domain name in the question), 10 bytes for the fixed-size fields (type, class, TTL, and resource length), and 19 bytes for the resource data (a domain name). Notice that the domain name in the resource data (`svr4.tuc.noao.edu.`) doesn't share a suffix with the domain name in the question (`34.13.252.140.in-addr.arpa.`) so a pointer can't be used.

14.3 Reversing the order means using the DNS first, and if that fails, trying to convert the argument as a dotted-decimal number. This means every time a dotted-decimal number is specified, the DNS is used, involving a name server. This is a waste of resources.

14.4 Section 4.2.2 of RFC 1035 specifies that a 2-byte length precedes the actual DNS message.

14.5 When a name server starts it normally reads the (possibly out of date) list of root servers from a disk file. It then tries to contact one of these root servers, requesting the name server records (a query type of NS) for the root domain. This returns the current up-to-date list of root servers. Minimally this requires one of the root server entries in the start-up disk file to be current.

14.6 The registration services of the InterNIC updates the root servers three times a week.

14.7 Since the resolver comes and goes, as applications come and go, if the system is configured to use multiple name servers and the resolver maintains no state, the resolver cannot keep track of the round-trip times to its various name servers. This can lead to timeouts for resolver queries that are too short, causing unnecessary retransmissions.

14.8 Sorting the A records should be done by the resolver, not the name server, since the resolver normally knows more than the server about the network topology of the client. (Newer releases of BIND provide for resolver sorting of A records.)

Chapter 15

15.1 TFTP requests sent to the broadcast address should be ignored. As stated in the Host Requirements RFC, responding to a broadcast request can create a significant security hole. A problem, however, is that not all implementations and APIs provide the destination address of a UDP datagram to the process that receives the datagram (Section 11.12). For this reason many TFTP servers don't enforce this restriction.

15.2 Unfortunately, the RFC says nothing about this block number wrap. Implementations should be able to transfer files up through 33,553,920 bytes (65535×512).

Many implementations fail when the size of the file exceeds 16,776,704 (32767×512) since they incorrectly maintain the block number as a signed 16-bit integer instead of an unsigned integer.

15.3 This simplifies coding a TFTP client to fit in read-only memory, because the server is the sender of the bootstrap files, so the server must implement the time-out and retransmission.

15.4 With its stop-and-wait protocol, TFTP can transfer a maximum of 512 bytes per client–server round trip. The maximum throughput of TFTP is then 512 bytes divided by the round-trip time between the client and server. On an Ethernet, assuming a round-trip time of 3 ms, the maximum throughput is around 170,000 bytes/sec.

Chapter 16

16.1 A router could forward an RARP request to some other host on one of the router's other attached networks, but sending a reply then becomes a problem. The router would also have to forward RARP replies.

BOOTP doesn't have this reply problem since the address to reply to is a normal IP address that the routers know how to forward anyway. The problem is that RARP uses only link-layer addresses, and routers don't normally know these values for hosts on other, nonattached, networks.

16.2 It could use its own hardware address, which should be unique, and which is set in the request and returned in the reply.

Chapter 17

17.1 All are mandatory except the UDP checksum. The IP checksum covers only the IP header, while the others start immediately after the IP header.

17.2 The source IP address, source port number, or protocol field might have been corrupted.

17.3 Many Internet applications use a carriage return and linefeed to mark the end of each application record. This is NVT ASCII coding (Section 26.4). An alternative technique is to prefix each record with a byte count, which is used by the DNS (Exercise 14.4) and Sun RPC (Section 29.2).

17.4 As we saw in Section 6.5, an ICMP error must return at least the first 8 bytes beyond the IP header of the IP datagram that caused the error. When TCP receives an ICMP error it needs to examine the two port numbers to determine which connection the error corresponds to, so the port numbers must be in the first 8 bytes of the TCP header.

17.5 There are options at the end of the TCP header, but there are no options in the UDP header.

Chapter 18

18.1 The ISN is a 32-bit counter that wraps around from 4,294,912,000 to 8,704 approximately 9.5 hours after the system was bootstrapped. After approximately another 9.5 hours it will wrap around to 17,408, then 26,112 after another 9.5 hours, and so on. Since the ISN starts at 1 when the system is bootstrapped, and since the lowest order digit cycles through 4, 8, 2, 6, and 0, the ISN should always be an odd number.

18.2 In the first case we used our `sock` program, and by default it transmits the Unix newline character as itself—the single ASCII character 012 (octal). In the second case we used the Telnet client and it converts the Unix newline into two ASCII characters—a carriage return (octal 015) followed by a linefeed (octal 012).

18.3 On a half-closed connection one end has sent a FIN and is waiting for either data or a FIN from the other end. A half-open connection is when one end crashes, unbeknown to the other end.

18.4 The 2MSL wait state is only entered for a connection that has gone through the ESTABLISHED state.

18.5 First, the daytime server does the active close of the TCP connection after writing the time and date to the client. This is indicated by the message printed by our `sock` program: "connection closed by peer." The client's end of the connection goes through the passive close states. This puts the socket pair in the TIME_WAIT state on the server, not the client.

Next, as shown in Section 18.6, most Berkeley-derived implementations allow a new connection request to arrive for a socket pair currently in the TIME_WAIT state, which is exactly what's happening here.

18.6 A reset is sent in response to the FIN, because the FIN arrived for a connection that was CLOSED.

18.7 The party that dials the number does the active open. The party whose telephone rings does the passive open. Simultaneous opens are not permitted, but a simultaneous close is OK.

18.8 We would only see ARP requests, not TCP SYN segments, but the ARP requests would have the same timing as in the figure.

18.9 The client is on the host `solaris` and the server is on the host `bsdi`. The client's ACK of the server's SYN is combined with the first data segment from the client (line 3). This is perfectly legal under the rules of TCP, although most implementations don't do this. Next, the client sends its FIN (line 4) before waiting for the ACK of its data. This allows the server to acknowledge both the data and the FIN in line 5.

This exchange (sending one segment of data from the client to the server) requires seven segments. The normal connection establishment and termination (Figure 18.13), along with a single data segment and its acknowledgment, requires nine segments.

18.10 First, the server's ACK of the client's FIN is normally not delayed (we discuss delayed ACKs in Section 19.3) but sent as soon as the FIN arrives. It takes the application a while to receive the EOF and tell its TCP to close its end of the connection. Second, the server that receives the FIN does not have to close its end of the connection on receiving the FIN from the client. As we saw in Section 18.5, data can still be sent.

18.11 If an arriving segment that generates an RST has an ACK field, the sequence number of the RST is the arriving ACK field. The ACK value of 1 in line 6 is relative to the ISN of 26368001 in line 2.

18.12 See [Crowcroft et al. 1992] for comments on layering.

18.13 Five queries are issued. Assume there are three packets to establish the connection, one for the query, one to ACK the query, one for the response, one to ACK the response, and four to terminate the connection. This means 11 packets per query, for a total of 55 packets. Using UDP reduces this to 10 packets.

This can be reduced to 10 packets per query if the ACK of the query is combined with the response (Section 19.3).

18.14 The limit is about 268 connections per second: the maximum number of TCP port numbers (65536 – 1024 = 64512, ignoring the well-known ports) divided by the TIME_WAIT state of 2MSL.

18.15 The duplicate FIN is acknowledged and the 2MSL timer is restarted.

18.16 The receipt of an RST while in the TIME_WAIT state causes the state to be prematurely terminated. This is called *TIME_WAIT assassination*. RFC 1337 [Braden 1992a] discusses this in detail and shows the potential problems. The simple fix proposed by this RFC is to ignore RST segments while in the TIME_WAIT state.

18.17 It's when the implementation does not support a half-close. Once the application causes a FIN to be sent, the application can no longer read from the connection.

18.18 No. Incoming data segments are demultiplexed using the source IP address, source port number, destination IP address, and destination port number. For incoming connection requests we saw in Section 18.11 that a TCP server can normally prevent connections from being accepted based on the destination IP address.

Chapter 19

19.1 Two application writes, followed by a read, cause a delay because the Nagle algorithm will probably be invoked. The first segment (with 8 bytes of data) is sent and its ACK is waited for before sending the 12 bytes of data. If the server implements delayed ACKs, there can be a delay of up to 200 ms (plus the RTT) before this ACK is received.

19.2 Assuming 5-byte CSLIP headers (IP and TCP) and 2 bytes of data, the RTT across the SLIP link for these segments is about 14.5 ms. We have to add to this the RTT

across the Ethernet (normally 5–10 ms), plus the routing time on sun and So yes, the observed times do appear correct.

19.3 In Figure 19.6 the time difference between segments 6 and 9 is 533 ms. In Figure 19.8 the time difference between segments 8 and 12 is 272 ms. (We measured the time for the F2 key, not the F1 key, since the first echo of the F1 key was lost in the second figure.)

Chapter 20

20.1 Byte number 0 is the SYN and byte number 8193 is the FIN. The SYN and FIN each occupy 1 byte in the sequence number space.

20.2 The first application write causes the first segment to be sent with the PUSH flag. Since BSD/386 always uses slow start, it waits for the first ACK before sending any more data. During this time the next three application writes occur, and the sending TCP buffers the data to send. The next three segments do not contain the PUSH flag since there is more data in the buffer to send. Eventually slow start catches up with the application writes and every application write causes a segment to be sent, and since that segment is the last one in the buffer, the PUSH flag is set.

20.3 Solving the bandwidth-delay equation for the capacity, it is 1,920 bytes for the first case, and 2,062 for the satellite case. It appears that the receiving TCP is only advertising a window of 2,048 bytes.

A window greater than 16,000 bytes should be able to saturate the satellite link.

20.4 No, because TCP can repacketize data after a timeout, as we'll see in Section 21.11.

20.5 Segment 15 is a window update sent automatically by the TCP module as a result of the application reading data, which causes the window to open. This is similar to segment 9 in that figure. Segment 16, however, is a result of the application closing its end of the connection.

20.6 This can cause the sender to inject packets into the network at a rate faster than the network can really handle. This is called *ACK compression* or *ACK smashing* [Mogul 1993, Sec. 15.8.13]. This reference indicates that ACK compression occurs on the Internet, although it rarely leads to congestion.

Chapter 21

21.1 The next timeout is for 48 seconds: $0 + 4 \times 12$. The factor of 4 is the next multiplier in the exponential backoff.

21.2 It appears SVR4 still uses the factor $2D$ instead of $4D$ in the calculation of RTO.

21.3 The stop-and-wait protocol used by TFTP is limited to 512 bytes of data per round trip. $32768/512 \times 1.5$ is 96 seconds.

21.4 Show four segments, numbered 1, 2, 3, and 4. Assume the order of receipt is 1, 3, 2, and 4. The ACKs generated by the receiver will be ACK 1 (a normal ACK), ACK 1 (a duplicate ACK when segment 3 is received out of order), ACK 3 when segment 2 is received (acknowledging both segments 2 and 3), and then ACK 4. Here one duplicate ACK is generated. If the order of receipt were 1, 3, 4, 2, two duplicate ACKs would be generated.

21.5 No, because the slope is still up and to the right, not downward.

21.6 See Figure E.1, p. 526.

21.7 In Figure 21.2 the segments contain 256 bytes of data, which takes approximately 250 ms to transfer across the 9600 bits/sec CSLIP link between `slip` and `bsdi`. Assuming the data segments are not queued somewhere between `bsdi` and `vangogh`, they arrive at `vangogh` about 250 ms apart. Since this exceeds the 200-ms delayed ACK timer, each segment is acknowledged when the next delayed ACK timer expires.

Chapter 22

22.1 The ACKs are probably all delayed on the host `bsdi`, because there is no reason to send them immediately. That's why the relative times have 0.170 and 0.370 as the fractional part. It also appears that the 200-ms timer on `bsdi` is running about 18 ms behind the same timer on `sun`.

22.2 The FIN flag, just like the SYN flag, occupies 1 byte in the sequence number space. The advertised window appears to be 1 byte smaller because TCP allows room for the 1 byte of sequence number space occupied by the FIN flag.

Chapter 23

23.1 It is usually simpler to invoke the keepalive option than explicitly coding application probes; the keepalive probes take less network bandwidth than application probes (since keepalive probes and answers contain no data); no probes are sent unless the connection is idle.

23.2 The keepalive option can cause a perfectly good connection to be dropped because of a temporary network outage; the probe interval (2 hours) is normally not configurable on an application basis;

Chapter 24

24.1 It means the sending TCP supports the window scale option, but doesn't need to scale its window for this connection. The other end (that receives this SYN) can then specify a window scale factor (that can be 0 or nonzero).

24.2 64000: the receive buffer size (128000) right shifted 1 bit. 55000: the receive buffer size (220000) right shifted 2 bits.

24.3 No. The problem is that acknowledgments are not reliably delivered (unless they're piggybacked with data) so a scale change appearing on an ACK could get lost.

24.4 $2^{32} \times 8 / 120$ equals 286 Mbits/sec, 2.86 times the FDDI data rate.

24.5 Each TCP would have to remember the last timestamp received on any connection from each host. Read Appendix B.2 of RFC 1323 for additional details.

24.6 The application must set the size of the receive buffer *before* establishing the connection with the other end, since the window scale option is sent in the initial SYN segment.

24.7 If the receiver ACKs every second data segment, the *throughput* is 1,118,881 bytes/sec. Using a window of 62 segments, with an ACK for every 31 segments, the value is 1,158,675.

24.8 With this option the timestamp echoed in the ACK is always from the segment that caused the ACK. There is no ambiguity about which retransmitted segment the ACK is for, but the other part of Karn's algorithm, dealing with the exponential backoff on retransmission, is still required.

24.9 The receiving TCP queues the data, but it cannot be passed to the application until the three-way handshake is complete: when the receiving TCP moves into the ESTABLISHED state.

24.10 Five segments are exchanged:

1. Client to server: SYN, data (request), and FIN. The server must queue the data as described in the previous exercise.
2. Server to client: SYN and ACK of client's SYN.
3. Client to server: ACK of server's SYN and client FIN (again). This causes the server to move to the ESTABLISHED state, and the queued data from segment 1 is passed to the server application.
4. Server to client: ACK of client FIN (which also acknowledges client data), data (server's reply), and server's FIN. This assumes that the SPT is short enough to allow this delayed ACK. When the client TCP receives this segment, the reply is passed to the client application, but the total time has been twice the RTT plus the SPT.
5. Client to server: ACK of server's FIN.

24.11 16,128 transactions per second (64,512 divided by 4).

24.12 The transaction time using T/TCP cannot be faster than the time required to exchange a UDP datagram between the two hosts. T/TCP should always take longer, since it still involves state processing that UDP doesn't do.

Chapter 25

25.1 If a system is running both a manager and agent, they are probably different processes. The manager listens on UDP port 162 for traps, and the agent listens on UDP port 161 for requests. If the same port were used for both traps and requests, separating the manager from the agent would be hard.

25.2 Refer to the section "Table Access" in Section 25.7.

Chapter 26

26.1 We expect segments 2, 4, and 9 from the server to be delayed. The time difference between segments 2 and 4 is 190.7 ms and the time difference between segments 2 and 9 is 400.7 ms.

All the ACKs from the client to the server appear to be delayed: segments 6, 11, 13, 15, 17, and 19. The time differences of the last five from segment 6 are 400.0, 600.0, 800.0, 1000.0, and 2.600 ms.

26.2 If one end of a connection is in TCP's urgent mode, then every time a segment is received, one is sent. This segment does not tell the receiver anything new (it is not acknowledging new data, for example), and it contains no data, it just reiterates that urgent mode has been entered.

26.3 There are only 512 of these reserved ports (512–1023), limiting a host to 512 Rlogin clients. The limit is normally less than 512 in real life, since some of the port numbers in this range are used as well-known ports by various servers, such as the Rlogin server.

TCP's limitation is that the socket pair defining a connection (the 4-tuple) must be unique. Since the Rlogin server always uses the same well-known port (513) multiple Rlogin clients on a given host can use the same reserved port only if they're connected to different server hosts. Rlogin clients, however, don't use this technique of trying to reuse reserved ports. If this technique were used, the theoretical limit is a maximum of 512 Rlogin clients at any one time that are all connected to the same server host.

Chapter 27

27.1 Theoretically the connection cannot be established while the socket pair is in the 2MSL wait on either end. Realistically, however, we saw in Section 18.6 that most Berkeley-derived implementations do accept a new SYN for a connection in the TIME_WAIT state.

27.2 These lines are not part of a server reply that begins with a 3-digit reply code, so they cannot be from the server.

Chapter 28

28.1 A *domain literal* is a dotted-decimal IP address within square brackets. For example: `mail rstevens@[140.252.1.54]`.

28.2 Six round trips: the HELO command, MAIL, RCPT, DATA, body of the message, and QUIT.

28.3 This is legal and is called *pipelining* [Rose 1993, Sec. 4.4.4]. Unfortunately there exist brain-damaged SMTP receiver implementations that clear their input buffer after each command is processed, causing this technique to fail. If this technique is used, naturally the client cannot discard the message until all the replies have been checked to verify that the message was accepted by the server.

28.4 Consider the first five network round trips from Exercise 28.2. Each is a small command (probably a single segment) that places little load on the network. If all five make it through to the server without retransmission, the congestion window could be six segments when the body is sent. If the body is large, the client could send the first six segments at once, which the network might not be able to handle.

28.5 Newer releases of BIND shuffle the MX records with the same value, as a form of load balancing.

Chapter 29

29.1 No, because `tcpdump` cannot distinguish an RPC request or reply from any other UDP datagram. The only time it interprets the contents of UDP datagrams as NFS packets is when the source or destination port number is 2049. Random RPC requests and replies can use an ephemeral port number on each end.

29.2 From Section 1.9 recall that a process must have superuser privileges to assign itself a port number less than 1024 (a well-known port). While this is OK for system-provided servers, such as the Telnet server, the FTP server, and the Port Mapper, we wouldn't want this restriction for all RPC servers.

29.3 Two concepts in this example are that the client ignores any server reply that doesn't have the XID that the client is waiting for, and UDP queues received datagrams (up to some limit) until the application reads the datagram. Also, the XID does not change on a timeout and retransmission, it changes only when another server procedure is called.

The events performed by the client stub are as follows: time 0: send request 1; time 4: time out and retransmit request 1; time 5: receive server's reply 1, return reply to application; time 5: send request 2; time 9: time out and retransmit request 2; time 10: receive server's reply 1, but ignore it since we're waiting for reply 2; time 11: receive server's reply 2, return reply to application.

The events at the server are as follows: time 0: receive request 1, start operation; time 5: send reply 1; time 5: receive request 1 (from client's retransmission at time 4), start operation; time 10: send reply 1; time 10: receive request 2 (from client's

transmission at time 5), start operation; time 11: send reply 2; time 11: receive request 2 (from client's retransmission at time 9), start operation; time 12: send reply 2. This final server reply is just queued by the client's UDP for the next receive done by the client. When the client reads it, the XID will be wrong, and the client will ignore it.

29.4 Changing the server's Ethernet card changes its physical address. Even though we noted in Section 4.7 that SVR4 does not send a gratuitous ARP on bootstrap, it still must send an ARP request for the physical address of `sun` before it can reply to its NFS requests. Since `sun` already has an ARP entry for `svr4`, it updates this entry with the sender's (new) hardware address from the ARP request.

29.5 The second of the client's block I/O daemons (reading at offset 73728) is out of sync with the first by about 0.74 seconds. That is, this second daemon times out 0.74 seconds after the first in lines 131–145. It appears the server never saw the request in line 167, but did see the request in line 168. The second block I/O daemon won't retransmit until 0.74 seconds after line 168, and in the mean time the first block I/O daemon continues issuing requests.

29.6 If TCP is used, and the TCP segment containing the server's reply is lost in the network, the server's TCP will time out and retransmit the reply when it doesn't receive an ACK from the client's TCP. Eventually the segment will arrive at the client's TCP. The difference here is that the two TCP modules do the timeout and retransmission, not the NFS client and server. (When UDP is used, the NFS client code performs the timeout and retransmission.) Therefore the NFS client never knows that the reply was lost and had to be retransmitted.

29.7 It is possible for the NFS server to obtain a different port number after the reboot. This would complicate the client, because it would have to know that the server crashed and contact the server's port mapper after the reboot to find the NFS server's new port number.

This scenario, where the server crashes and reboots and a server RPC application obtains a new ephemeral port, can happen to any RPC application that doesn't use a well-known port.

29.8 No. The NFS client can reuse the same local, reserved port number for different servers. TCP requires the 4-tuple of local IP address, local port, foreign IP address, and foreign port to be unique, and the foreign IP address is different for each server host.

Chapter 30

30.1 Type `whois "net 88"`. Class A network IDs 64 through 95 are reserved.

30.2 Type `whois whitehouse-dom`. Either the `host` command or `nslookup` can query the DNS.

30.3 No, `xscope` can run on a different host from the server. If the hosts are different, then `xscope` can also use TCP port 6000 for its incoming connection.

Appendix E

Configurable Options

We've seen many features of TCP/IP that we've had to describe with the qualifier "it depends on the configuration." Typical examples are whether or not UDP checksums are enabled (Section 11.3), whether destination IP addresses with the same network ID but a different subnet ID are local or nonlocal (Section 18.4), and whether directed broadcasts are forwarded or not (Section 12.3). Indeed, many operating characteristics of a given TCP/IP implementation can be modified by the system administrator.

This appendix lists some of the configurable options for the various TCP/IP implementations that have been used throughout the text. As you might expect, every vendor does things differently from all others. Nevertheless, this appendix gives an idea of the types of parameters different implementations allow one to modify. A few options that are highly implementation specific, such as the low-water mark for the memory buffer pool, are not described.

> *These variables are described for informational purposes only. Their names, default values, or interpretation can change from one release to the next. Always check your vendor's documentation (or bug them for adequate documentation) for the final word on these variables.*

This appendix does not cover the initialization that takes place every time the system is bootstrapped: the initialization of each network interface using `ifconfig` (setting the IP address, the subnet mask, etc.), entering static routes into the routing table, and the like. Instead, this appendix focuses on the configuration options that affect how TCP/IP operates.

E.1 BSD/386 Version 1.0

This system is an example of the "classical" BSD configuration that has been used since 4.2BSD. Since the source code is distributed with the system, configuration options are specified by the administrator, and the kernel is recompiled. There are two types of options: constants that are defined in the kernel configuration file (see the `config`(8) manual page), and variable initializations in various C source files. Brave and knowledgeable administrators can also change the values of these C variables in either the running kernel or the kernel's disk image, using a debugger, to avoid rebuilding the kernel.

Here are the constants that can be changed in the kernel's configuration file.

`IPFORWARDING`
The value of this constant initializes the kernel variable `ipforwarding`. If 0 (default), IP datagrams are not forwarded. If 1, forwarding is always enabled.

`GATEWAY`
If defined, causes `IPFORWARDING` to be set to 1. Additionally, defining this constant causes certain system tables (the ARP cache and the routing table) to be larger.

`SUBNETSARELOCAL`
The value of this constant initializes the kernel variable `subnetsarelocal`. If 1 (default), a destination IP address with the same network ID as the sending host but a different subnet ID is considered local. If 0, only destination IP addresses on an attached subnet are considered local. This is summarized in Figure E.1.

Network IDs	Subnet IDs	`subnetsarelocal`		Comment
		1	0	
same	same	local	local	always local
same	different	local	nonlocal	depends on configuration
different		nonlocal	nonlocal	always nonlocal

Figure E.1 Interpretation of `subnetsarelocal` kernel variable.

This affects the MSS selected by TCP. When sending to local destinations, TCP chooses the MSS based on the MTU of the outgoing interface. When sending to nonlocal destinations, TCP uses the variable `tcp_mssdflt` as the MSS.

`IPSENDREDIRECTS`
The value of this constant initializes the kernel variable `ipsendredirects`. If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

`DIRECTED_BROADCAST`
If 1 (default), received datagrams whose destination address is the directed broadcast address of an attached interface are forwarded as a link-layer broadcast. If 0, these datagrams are silently discarded.

The following variables can also be modified. These variables are spread throughout different files in the `/usr/src/sys/netinet` directory.

`tcprexmtthresh`	The number of consecutive ACKs that triggers the fast retransmit and fast recovery algorithm. The default value is 3.
`tcp_ttl`	The default value for the TTL field for TCP segments. Default value is 60.
`tcp_mssdflt`	The default TCP MSS for nonlocal destinations. Default value is 512.
`tcp_keepidle`	Number of 500-ms clock ticks before sending a keepalive probe. Default value is 14400 (2 hours).
`tcp_keepintvl`	Number of 500-ms clock ticks between successive keepalive probes, when no response is received. Default value is 150 (75 seconds).
`tcp_sendspace`	The default size of the TCP send buffer. Default value is 4096.
`tcp_recvspace`	The default size of the TCP receive buffer. This affects the window size that is offered. Default value is 4096.
`udpcksum`	If nonzero, UDP checksums are calculated for outgoing UDP datagrams, and incoming UDP datagrams containing nonzero checksums have their checksum verified. If 0, outgoing UDP datagrams do not contain a checksum, and no checksum verification is performed on incoming UDP datagrams, even if the sender calculated a checksum. Default is 1.
`udp_ttl`	The default value for the TTL field in UDP datagrams. Default value is 30.
`udp_sendspace`	The default size of the UDP send buffer. Defines the maximum UDP datagram that can be sent. Default is 9216.
`udp_recvspace`	The default size of the UDP receive buffer. The default is 41600, allowing for 40 1024-byte datagrams.

E.2 SunOS 4.1.3

The method used with SunOS 4.1.3 is similar to what we saw with BSD/386. Since most of the kernel sources are not distributed, all the C variable initializations are contained in a single C source file that is provided.

The administrator's kernel configuration file (see the `config`(8) manual page) can define the following variables. After modifying your configuration file, a new kernel must be made and rebooted.

IPFORWARDING
The value of this constant initializes the kernel variable `ip_forwarding`. If –1, IP datagrams are never forwarded. Furthermore, the variable is never changed. If 0 (default), IP datagrams are not forwarded, but the variable's value is changed to 1 if multiple interfaces are up. If 1, forwarding is always enabled.

SUBNETSARELOCAL
The value of the kernel variable `ip_subnetsarelocal` is initialized from this constant. If 1 (default), a destination IP address with the same network ID as the sending host but a different subnet ID is considered local. If 0, only destination IP addresses on an attached subnet are considered local. This is summarized in Figure E.1. When sending to local destinations, TCP chooses the MSS based on the MTU of the outgoing interface. When sending to nonlocal destinations, TCP uses the variable `tcp_default_mss`.

IPSENDREDIRECTS
The value of this constant initializes the kernel variable `ip_sendredirects`. If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

DIRECTED_BROADCAST
The value of this constant initializes the kernel variable `ip_dirbroadcast`. If 1 (default), received datagrams whose destination address is the directed broadcast address of an attached interface are forwarded as a link-layer broadcast. If 0, these datagrams are silently discarded.

The file `/usr/kvm/sys/netinet/in_proto.c` defines the following variables that can be changed. Once these variables are changed, a new kernel must be made and rebooted.

`tcp_default_mss`	The default TCP MSS for nonlocal destinations. Default value is 512.
`tcp_sendspace`	The default size of the TCP send buffer. Default value is 4096.
`tcp_recvspace`	The default size of the TCP receive buffer. This affects the window size that is offered. Default value is 4096.
`tcp_keeplen`	A keepalive probe to a 4.2BSD host must contain a single byte of data to get a response. Set the variable to 1 for compatibility with these older implementations. Default value is 1.
`tcp_ttl`	The default value for the TTL field for TCP segments. Default value is 60.
`tcp_nodelack`	If nonzero, ACKs are not delayed. Default value is 0.
`tcp_keepidle`	Number of 500-ms clock ticks before sending a keepalive probe. Default value is 14400 (2 hours).

`tcp_keepintvl` Number of 500-ms clock ticks between successive keepalive probes, when no response is received. Default value is 150 (75 seconds).

`udp_cksum` If nonzero, UDP checksums are calculated for outgoing UDP datagrams, and incoming UDP datagrams containing nonzero checksums have their checksum verified. If 0, outgoing UDP datagrams do not contain a checksum, and no checksum verification is performed on incoming UDP datagrams, even if the sender calculated a checksum. Default is 0.

`udp_ttl` The default value for the TTL field in UDP datagrams. Default value is 60.

`udp_sendspace` The default size of the UDP send buffer. Defines the maximum UDP datagram that can be sent. Default is 9000.

`udp_recvspace` The default size of the UDP receive buffer. The default is 18000, allowing for two 9000-byte datagrams.

E.3 System V Release 4

The TCP/IP configuration of SVR4 is similar to the previous two systems, but fewer options are available. In the file `/etc/conf/pack.d/ip/space.c` two constants can be defined, and the kernel must then be rebuilt and rebooted.

`IPFORWARDING`
The value of this constant initializes the kernel variable `ipforwarding`. If 0 (default), IP datagrams are not forwarded. If 1, forwarding is always enabled.

`IPSENDREDIRECTS`
The value of this constant initializes the kernel variable `ipsendredirects`. If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

Many of the variables that we've described in the previous two sections are defined in the kernel, but one must patch the kernel to modify them. For example, there is a variable named `tcp_keepidle` with a value of 14400.

E.4 Solaris 2.2

Solaris 2.2 is typical of the newer Unix systems that provide a program for the administrator to run to change the configuration options of the TCP/IP system. This allows reconfiguration without having to modify source files and rebuild a kernel.

The configuration program is `ndd`(1). We can run the program to see what parameters we can examine or modify in the UDP module:

```
solaris % ndd /dev/udp \?
udp_wroff_extra                 (read and write)
udp_def_ttl                     (read and write)
udp_first_anon_port             (read and write)
udp_trust_optlen                (read and write)
udp_do_checksum                 (read and write)
udp_status                      (read only)
```

There are five modules we can specify: `/dev/ip`, `/dev/icmp`, `/dev/arp`, `/dev/udp`, and `/dev/tcp`. The question mark argument (which we have to prevent the shell from interpreting by preceding it with a backslash) tells the program to list all the parameters for that module. An example that queries the value of a variable is:

```
solaris % ndd /dev/tcp tcp_mss_def
536
```

To change the value of a variable we need superuser privilege and type:

```
solaris # ndd -set /dev/ip ip_forwarding 0
```

These variables can be divided into three categories:

1. Configuration variables that a system administrator can change (e.g., `ip_forwarding`).
2. Status variables that can only be displayed (e.g., the ARP cache). Normally this information is provided in an easier to understand format by the commands `ifconfig`, `netstat`, and `arp`.
3. Debugging variables intended for those with kernel source code. Enabling some of these generates kernel debug output at runtime, which can degrade performance.

We now describe the parameters in each module. All parameters are read–write, unless marked "(Read only)." The read-only parameters are the status variables from case 2 above. We also mark the "(Debug)" variables from case 3. Unless otherwise noted, all the timing variables are specified in milliseconds, which differs from the other systems that normally specify times as some number of 500-ms clock ticks.

/dev/ip

`ip_cksum_choice`
(Debug) Selects between two independent implementations of the IP checksum algorithm.

`ip_debug`
(Debug) Enables printing of debug output by the kernel, if greater than 0. Larger values generate more output. Default is 0.

`ip_def_ttl`
Default TTL for outgoing IP datagrams, if not specified by transport layer. Default is 255.

`ip_forward_directed_broadcasts`
If 1 (default), received datagrams whose destination address is the directed broadcast address of an attached interface are forwarded as a link-layer broadcast. If 0, these datagrams are silently discarded.

`ip_forward_src_routed`
If 1 (default), received datagrams containing a source route option are forwarded. If 0, these datagrams are discarded.

`ip_forwarding`
Specifies whether the system forwards incoming IP datagrams: 0 means never forward, 1 means always forward, and 2 (default) means only forward when two or more interfaces are up.

`ip_icmp_return_data_bytes`
The number of bytes of data beyond the IP header that are returned in an ICMP error. Default is 64.

`ip_ignore_delete_time`
(Debug) Minimum lifetime of an IP routing table entry (IRE). Default is 30 seconds. (This parameter is in seconds, not milliseconds.)

`ip_ill_status`
(Read only) Displays the status of each IP lower layer data structure. There is one lower layer structure for each interface.

`ip_ipif_status`
(Read only) Displays the status of each IP interface data structure (IP address, subnet mask, etc.). There is one of these structures for each interface.

`ip_ire_cleanup_interval`
(Debug) The interval at which the IP routing table entries are scanned for possible deletions. Default is 30000 ms (30 seconds).

`ip_ire_flush_interval`
The interval at which ARP information in unconditionally flushed from the IP routing table. Default is 1200000 ms (20 minutes).

`ip_ire_pathmtu_interval`
The interval at which the path MTU discovery algorithm tries to increase the MTU. Default is 30000 ms (30 seconds).

`ip_ire_redirect_interval`
The interval at which IP routing table entries that are from ICMP redirects are deleted. Default is 60000 ms (60 seconds).

`ip_ire_status`
(Read only) Displays all the IP routing table entries.

`ip_local_cksum`
If 0 (default), IP does not calculate the IP checksum or the higher layer protocol checksum (i.e., TCP, UDP, ICMP, or IGMP) for datagrams sent or received through the loopback interface. If 1, these checksums are calculated.

`ip_mrtdebug`
(Debug) Enables printing of debug output concerning multicast routing by the kernel, if 1. Default is 0.

`ip_path_mtu_discovery`
If 1 (default), path MTU discovery is performed by IP. If 0, IP never sets the "don't fragment" bit in outgoing datagrams.

`ip_respond_to_address_mask`
If 0 (default), the host does not respond to ICMP address mask requests. If 1, it does respond.

`ip_respond_to_echo_broadcast`
If 1 (default), the host responds to ICMP echo requests that are sent to a broadcast address. If 0, it does not respond.

`ip_respond_to_timestamp`
If 0 (default), the host does not respond to ICMP timestamp requests. If 1, the host responds.

`ip_respond_to_timestamp_broadcast`
If 0 (default), the host does not respond to ICMP timestamp requests that are sent to a broadcast address. If 1, it does respond.

`ip_rput_pullups`
(Debug) Count of number of buffers from the network interface driver that needed to be pulled up to access the full IP header. Initialized to 0 at bootstrap time, and can be reset to 0.

`ip_send_redirects`
If 1 (default), the host sends ICMP redirects when acting as a router. If 0, these are not sent.

`ip_send_source_quench`
If 1 (default), the host generates ICMP source quench errors when incoming datagrams are discarded. If 0, these are not generated.

`ip_wroff_extra`
(Debug) Number of bytes of extra space to allocate in buffers for IP headers. Default is 32.

/dev/icmp

`icmp_bsd_compat`
(Debug) If 1 (default), the length field in the IP header of received datagrams is adjusted to exclude the length of the IP header. This is compatible with Berkeley-derived implementations and is for applications reading raw IP or raw ICMP packets. If 0, the length field is not changed.

`icmp_def_ttl`
The default TTL for outgoing ICMP messages. Default is 255.

`icmp_wroff_extra`
(Debug) Number of bytes of extra space to allocate in buffers for IP options and data-link headers. Default is 32.

/dev/arp

`arp_cache_report`
(Read only) The ARP cache.

`arp_cleanup_interval`
The interval after which ARP entries are discarded from ARP's cache. Default is 300000 ms (5 minutes). (IP maintains its own cache of completed ARP translations; see `ip_ire_flush_interval`.)

`arp_debug`
(Debug) If 1, enables printing of debug output by the ARP driver. Default is 0.

/dev/udp

`udp_def_ttl`
The default TTL for outgoing UDP datagrams. Default value is 255.

`udp_do_checksum`
If 1 (default), UDP checksums are calculated for outgoing UDP datagrams. If 0, outgoing UDP datagrams do not contain a checksum. (Unlike most other implementations, this UDP checksum flag does not affect incoming datagrams. If a received datagram has a nonzero checksum, it is always verified.)

`udp_largest_anon_port`
Largest port number to allocate for UDP ephemeral ports. Default is 65535.

`udp_smallest_anon_port`
Starting port number to allocate for UDP ephemeral ports. Default is 32768.

`udp_smallest_nonpriv_port`
A process requires superuser privilege to assign itself a port number less than this. Default is 1024.

`udp_status`
(Read only) The status of all local UDP end points: local IP address and port, foreign IP address and port.

`udp_trust_optlen`
(Debug) No longer used.

`udp_wroff_extra`
(Debug) Number of bytes of extra space to allocate in buffers for IP options and data-link headers. Default is 32.

/dev/tcp

`tcp_close_wait_interval`
The 2MSL value: the time spent in the TIME_WAIT state. Default is 240000 ms (4 minutes).

`tcp_conn_grace_period`
(Debug) Additional time added to the timer interval when sending a SYN. Default is 500 ms.

`tcp_conn_req_max`
The maximum number of pending connection requests queued for any listening end point. Default is 5.

`tcp_cwnd_max`
The maximum value of the congestion window. Default is 32768.

`tcp_debug`
(Debug) If 1, enables printing of debug output by TCP. Default is 0.

`tcp_deferred_ack_interval`
The time to wait before sending a delayed ACK. Default is 50 ms.

`tcp_dupack_fast_retransmit`
The number of consecutive duplicate ACKs that triggers the fast retransmit, fast recovery algorithm. Default is 3.

`tcp_eager_listeners`
(Debug) If 1 (default), TCP completes the three-way handshake before returning a new connection to an application with a pending passive open. This is the way most TCP implementations operate. If 0, TCP passes an incoming connection request (received SYN) to the application, and does not complete the three-way handshake until the application accepts the connection. (Setting this to 0 might break many existing applications.)

`tcp_ignore_path_mtu`
(Debug) If 1, path MTU discovery ignores received ICMP fragmentation needed messages. If 0 (default), path MTU discovery is enabled for TCP.

`tcp_ip_abort_cinterval`
The total retransmit timeout value when TCP is performing an active open. Default is 240000 ms (4 minutes).

`tcp_ip_abort_interval`
The total retransmit timeout value for a TCP connection after it is established. Default is 120000 ms (2 minutes).

`tcp_ip_notify_cinterval`
The timeout value when TCP is performing an active open after which TCP notifies IP to find a new route. Default is 10000 ms (10 seconds).

`tcp_ip_notify_interval`
The timeout value for an established connection after which TCP notifies IP to find a new route. Default is 10000 ms (10 seconds).

`tcp_ip_ttl`
The TTL to use for outgoing TCP segments. Default is 255.

`tcp_keepalive_interval`
The time that a connection must be idle before a keepalive probe is sent. Default is 7200000 ms (2 hours).

`tcp_largest_anon_port`
Largest port number to allocate for TCP ephemeral ports. Default is 65535.

`tcp_maxpsz_multiplier`
(Debug) Specifies the multiple of the MSS into which the stream head packetizes the application's write data. Default is 1.

`tcp_mss_def`
Default MSS for nonlocal destinations. Default is 536.

`tcp_mss_max`
The maximum MSS. Default is 65495.

`tcp_mss_min`
The minimum MSS. Default is 1.

`tcp_naglim_def`
(Debug) Maximum value of the per-connection Nagle algorithm threshold. Default is 65535. The per-connection value starts out as the minimum of the MSS or this value. The per-connection value is set to 1 by the `TCP_NODELAY` socket option, which disables the Nagle algorithm.

`tcp_old_urp_interpretation`
(Debug) If 1 (default), the older (but more common) BSD interpretation of the urgent pointer is used: it points 1 byte beyond the last byte of urgent data. If 0, the Host Requirements RFC interpretation is used: it points to the last byte of urgent data.

`tcp_rcv_push_wait`
(Debug) Maximum number of bytes received without the PUSH flag set before the data is passed to the application. Default is 16384.

`tcp_rexmit_interval_initial`
(Debug) Initial retransmit timeout interval. Default is 500 ms.

`tcp_rexmit_interval_max`
(Debug) Maximum retransmit timeout interval. Default is 60000 ms (60 seconds).

`tcp_rexmit_interval_min`
(Debug) Minimum retransmit timeout interval. Default is 200 ms.

`tcp_rwin_credit_pct`
(Debug) Percentage of receive window that must be buffered before flow control is checked on every received segment. Default is 50%.

`tcp_smallest_anon_port`
Starting port number to allocate for TCP ephemeral ports. Default is 32768.

`tcp_smallest_nonpriv_port`
A process requires superuser privilege to assign itself a port number less than this. Default is 1024.

`tcp_snd_lowat_fraction`
(Debug) If nonzero, the send buffer low-water mark is the send buffer size divided by this value. Default is 0 (disabled).

`tcp_status`
(Read only) Information on all TCP connections.

`tcp_sth_rcv_hiwat`
(Debug) If nonzero, the value to set the stream head high-water mark to. Default is 0.

`tcp_sth_rcv_lowat`
(Debug) If nonzero, the value to set the stream head low-water mark to. Default is 0.

`tcp_wroff_xtra`
(Debug) Number of bytes of extra space to allocate in buffers for IP options and data-link headers. Default is 32.

E.5 AIX 3.2.2

AIX 3.2.2 allows network options to be set at runtime using the `no` command. It can display the value of an option, set the value of an option, or set an option value back to its default. For example, to display an option we type:

```
aix % no -o udp_ttl
udp_ttl = 30
```

The following options can be modified.

`arpt_killc`
The time (in minutes) before an inactive completed ARP entry is removed. Default is 20.

`ipforwarding`
If 1 (default), IP datagrams are always forwarded. If 0, forwarding is disabled.

`ipfragttl`
The time to live (in seconds) for IP fragments awaiting reassembly. Default is 60.

`ipsendredirects`
If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

`loop_check_sum`
If 1 (default), the IP checksum is calculated for datagrams sent through the loopback interface. If 0, this checksum is not calculated.

`nonlocsrcroute`
If 1 (default), received datagrams containing a source route option are forwarded. If 0, these datagrams are discarded.

`subnetsarelocal`
If 1 (default), a destination IP address with the same network ID as the sending host but a different subnet ID is considered local. If 0, only destination IP addresses on an attached subnet are considered local. This is summarized in Figure E.1. When sending to local destinations, TCP chooses the MSS based on the MTU of the outgoing interface. When sending to nonlocal destinations, TCP uses the default (536) as the MSS.

`tcp_keepidle`
Number of 500-ms clock ticks before sending a keepalive probe. Default value is 14400 (2 hours).

`tcp_keepintvl`
Number of 500-ms clock ticks between successive keepalive probes, when no response is received. Default value is 150 (75 seconds).

`tcp_recvspace`
The default size of the TCP receive buffer. This affects the window size that is offered. Default value is 16384.

`tcp_sendspace`
The default size of the TCP send buffer. Default value is 16384.

`tcp_ttl`
The default value for the TTL field for TCP segments. Default value is 60.

`udp_recvspace`
The default size of the UDP receive buffer. The default is 41600, allowing for 40 1024-byte datagrams.

`udp_sendspace`
The default size of the UDP send buffer. Defines the maximum UDP datagram that can be sent. Default is 9216.

`udp_ttl`
The default value for the TTL field in UDP datagrams. Default value is 30.

E.6 4.4BSD

4.4BSD is the first of the Berkeley releases to provide dynamic configuration for numerous kernel parameters. The `sysctl`(8) command is used. The names for the parameters were chosen to look like MIB names from SNMP. To examine a parameter we type:

```
vangogh % sysctl net.inet.ip.forwarding
net.inet.ip.forwarding = 1
```

To change a parameter we need superuser privilege and then type:

```
vangogh # sysctl -w net.inet.ip.ttl=128
```

The following parameters can be changed.

`net.inet.ip.forwarding`
If 0 (default), IP datagrams are not forwarded. If 1, forwarding is enabled.

`net.inet.ip.redirect`
If 1 (default), the host will send ICMP redirects when forwarding IP datagrams. If 0, ICMP redirects are not sent.

`net.inet.ip.ttl`
The default TTL for both TCP and UDP. The default is 64.

`net.inet.icmp.maskrepl`
If 0 (default), the host does not respond to ICMP address mask requests. If 1, it does respond.

`net.inet.udp.checksum`
If 1 (default), UDP checksums are calculated for outgoing UDP datagrams, and incoming UDP datagrams containing nonzero checksums have their checksum verified. If 0, outgoing UDP datagrams do not contain a checksum, and no checksum verification is performed on incoming UDP datagrams, even if the sender calculated a checksum.

Additionally, numerous variables that we've described earlier in this appendix are scattered among various source files (`tcp_keepidle`, `subnetsarelocal`, etc.) and can be modified.

Appendix F

Source Code Availability

This book uses many publicly available software packages. This appendix provides additional details on how to obtain this software.

The technique used to obtain this software is called *anonymous FTP*, where FTP is the standard Internet File Transfer Protocol (Chapter 27). Section 27.3 shows an example of anonymous FTP. For a background on Internet resources in general, and specifically anonymous FTP, refer to any of the recently available books on the Internet, such as [LaQuey 1993] or [Krol 1992].

The hosts listed here are believed to be the primary site where the package is available. There may be many other sites where the software is also available. The Internet Archie service can locate additional versions. Also, the versions listed below are the ones used for the examples in the text.

Newer versions may have been released by the time you read this.

You should use the FTP `dir` command to see if newer versions exist on that specified host.

This appendix is ordered by the chapter or section number where the resource was used in this text.

RFCs (Section 1.11)

Section 1.11 provides the electronic mail address to send a request to. The reply details numerous sites from which the RFCs can be obtained using either e-mail or anonymous FTP.

Remember that the starting place is to obtain the current index and look up the RFC that you want in the index. This entry tells you if that RFC has been obsoleted or updated by a newer RFC.

BSD Net/2 Source Code (Section 1.14)

The BSD Net/2 source code, which includes the kernel implementation of the TCP/IP protocols, along with the standard utilities (Telnet client and server, FTP client and server, etc.), is available from `ftp.uu.net` in the directory tree starting at `systems/unix/bsd-sources`.

SLIP (Section 2.4)

The version of SLIP used in this text is available from `ftp.ee.lbl.gov`. The filename begins with `cslip`, since it supports compressed SLIP (Section 2.5).

`icmpaddrmask` Program (Section 6.3)

Refer to the final entry of this section.

`icmptime` Program (Section 6.4)

Refer to the final entry of this section.

`ping` Program (Chapter 7)

The BSD version of `ping` normally has more options and features than the version supplied by many vendors. The host `ftp.uu.net` contains the latest BSD version in the file `systems/unix/bsd-sources/sbin/ping`.

`traceroute` Program (Chapter 8)

The `traceroute` program is available from `ftp.ee.lbl.gov`. Refer to the final entry of this section for the version used in Section 8.5 that allows loose and strict source routing.

Router Discovery Daemon (Section 9.6)

A program is available that provides host support and router support for the router discovery messages. The host is `gregorio.stanford.edu` and the file is `gw-discovery/nordmark-rdisc.tar`. The program was written by Sun Microsystems and made publicly available.

`gated` Daemon (Section 10.3)

The `gated` routing daemon, mentioned in Section 10.3, is available from the host `gated.cornell.edu`.

`traceroute.pmtu` Program (Section 11.7)

Refer to the final entry of this section.

IP Multicasting Software (Chapter 13)

The modifications required to support IP multicasting for SunOS 4.x and Ultrix are available from `gregorio.stanford.edu` in the directory `vmtp-ip`. This directory also contains the source code modifications required to implement IP multicasting in a Berkeley Unix system.

BIND Name Server (Chapter 14)

The BIND name server, the `named` daemon, is available from the host `ftp.uu.net` in the file `networking/ip/dns/bind/bind.4.8.3.tar.Z`.

A newer version, 4.9, is available from `gatekeeper.dec.com` in the directory `pub/BSD/bind/4.9`.

`host` Program (Chapter 14)

The `host` program is available from the host `nikhefh.nikhef.nl` in the file `host.tar.Z`.

`dig` and `doc` Programs (Chapter 14)

The `dig` and `doc` programs mentioned in Chapter 14 are available from the host `isi.edu` in the files `dig.2.0.tar.Z` and `doc.2.0.tar.Z`.

BOOTP Server (Chapter 16)

Various versions of the commonly used Unix BOOTP server are available from the host `lancaster.andrew.cmu.edu`, in the `pub` directory.

TCP High-Speed Extensions (Chapter 24)

A publicly available source code implementation of the TCP window scale option, timestamp option, and PAWS algorithm is available as a set of patches to the BSD Net/2 release from the host `uxc.cso.uiuc.edu` in the file `pub/tcplw.shar.Z`.

ISODE SNMP Manager and Agent (Chapter 25)

The SNMP manager and agent described in Section 25.7 are part of the ISODE 8.0 distribution. This is available from many FTP archive sites, such as `ftp.uu.net` in the `networking/osi/isode` directory.

MIME Software and Examples (Section 28.4)

A program named MetaMail that provides MIME capabilities for many different user agents is available on the host `thumper.bellcore.com` in the `pub/nsb` directory. Also in this directory is additional information on MIME.

Sun RPC (Section 29.2)

A version of the RPC 4.0 sources (which use the sockets API) is available from the host `ftp.uu.net` in the `systems/sun/sextape/rpc4.0` directory. A version of the TI-RPC sources (which use the TLI API) is available from the host `ftp.uu.net` in the `networking/rpc` directory.

Sun NFS (Chapter 29)

A publicly available implementation of an NFS client and server is provided as part of the BSD Net/2 Source Code described earlier in this appendix.

`tcpdump` Program (Appendix A)

The version of `tcpdump` used in this text is from the host `ftp.ee.lbl.gov` in the file `tcpdump-2.2.1.tar.Z`.

BSD Packet Filter (Section A.1)

The BSD packet filter is part of the `tcpdump` distribution.

`sock` Program (Appendix C)

Refer to the final entry of this section.

`ttcp` Program

(This program was not used in the text, but is a useful tool of which readers should be aware.) `ttcp` is a benchmarking tool for measuring TCP and UDP performance between two systems. It was created at the U.S. Army Ballistics Research Lab (BRL) and is in the public domain. Copies are available from many anonymous FTP sites but an enhanced version is available from `ftp.sgi.com` in the directory `sgi/src/ttcp`.

Author-Written Software

The author-written software used in the book is available from the host `ftp.uu.net` in the file `published/books/stevens.tcpipiv1.tar.Z`.

Bibliography

All the RFCs are available at no charge through electronic mail or using anonymous FTP across the Internet as described in Section 1.11.

Albitz, P., and Liu, C. 1992. *DNS and BIND.* O'Reilly & Associates, Sebastopol, Calif.

Lots of details on the administrative tasks required to configure and run a name server.

Alexander, S., and Droms, R. 1993. "DHCP Options and BOOTP Vendor Extensions," RFC 1533, 30 pages (Oct.).

Almquist, P. 1992. "Type of Service in the Internet Protocol Suite," RFC 1349, 28 pages (July).

How to use the type-of-service field in the IP header.

Almquist, P., ed. 1993. "Requirements for IP Routers," Internet Draft (Mar.).

This is the draft of an RFC to replace RFC 1009 [Braden and Postel 1987]. The new RFC will probably appear in four volumes. Volume 1: Internet architecture, terminology, and general considerations. Volume 2: link layer, internet layer, transport layer, and application layer. Volume 3: forwarding and routing protocols. Volume 4: operations and maintenance, and network management.

This draft is available via anonymous FTP from the host `jessica.stanford.edu` in the directory `rreq`. Ignore this draft once the final RFC is published.

Bellovin, S. M. 1993. Private Communication.

Bhide, A., Elnozahy, E. N., and Morgan, S. P. 1991. "A Highly Available Network File Server," *Proceedings of the 1991 Winter USENIX Conference*, pp. 199–205, Dallas, Tex.

Describes a use of gratuitous ARP (Section 4.7).

Borenstein, N., and Freed, N. 1993. "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," RFC 1521, 81 pages (Sept.).

This RFC obsoletes the earlier RFC 1341. Appendix H of this RFC lists the differences from RFC 1341.

Borman, D. A., ed. 1990. "Telnet Linemode Option," RFC 1184, 23 pages (Oct.).

Borman, D. A. 1991. "IP Bandwidth Limits," Message-ID <91011437.AA17276@berserkly.cray.com>, Usenet, comp.protocols.tcp-ip Newsgroup (Jan.).

States the three practical limits on TCP performance that we listed at the end of Section 24.8.

Borman, D. A. 1992. "TCP/IP Performance at Cray Research," *Proceedings of the Twenty-third Internet Engineering Task Force*, pp. 492–493 (Mar.), San Diego Supercomputer Center, San Diego, Calif.

Borman, D. A., ed. 1993a. "Telnet Environment Option," RFC 1408, 7 pages (Jan.).

The Telnet option for passing environment variables from the client to the server.

Borman, D. A. 1993b. "A Practical Perspective on Host Networking," in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 309–367. Addison-Wesley, Reading, Mass.

A practical look at the Host Requirements RFCs (1122 and 1123).

Braden, R. T., ed. 1989a. "Requirements for Internet Hosts—Communication Layers," RFC 1122, 116 pages (Oct.).

The first half of the Host Requirements RFC. This half covers the link layer, IP, TCP, and UDP.

Braden, R. T., ed. 1989b. "Requirements for Internet Hosts—Application and Support," RFC 1123, 98 pages (Oct.).

The second half of the Host Requirements RFC. This half covers Telnet, FTP, TFTP, SMTP, and the DNS.

Braden, R. T. 1989c. "Perspective on the Host Requirements RFCs," RFC 1127, 20 pages (Oct.).

An informal summary of the discussions and conclusions of the IETF working group that developed the Host Requirements RFC.

Braden, R. T. 1992a. "TIME-WAIT Assassination Hazards in TCP," RFC 1337, 11 pages (May).

Shows how the receipt of an RST while in the TIME_WAIT state can lead to problems.

Braden, R. T. 1992b. "Extending TCP for Transactions—Concepts," RFC 1379, 38 pages (Nov.).

The concepts and history behind the development of T/TCP.

Braden, R. T. 1992c. "Extending TCP for Transactions—Functional Specification," Internet Draft, 32 pages (Dec.).

The functional specification and a discussion of the implementation issues in T/ TCP.

Braden, R. T., Borman, D. A., and Partridge, C. 1988. "Computing the Internet Checksum," RFC 1071, 24 pages (Sept.).

Provides techniques and algorithms for calculating the checksum used by IP, ICMP, IGMP, UDP, and TCP.

Braden, R. T., and Postel, J. B. 1987. "Requirements for Internet Gateways," RFC 1009, 55 pages (June).

The equivalent of the Host Requirements RFC for routers. This RFC is being replaced; see [Almquist 1993].

Caceres, R., Danzig, P. B., Jamin, S., and Mitzel, D. J. 1991. "Characteristics of Wide-Area TCP/IP Conversations," *Computer Communication Review*, vol. 21, no. 4, pp. 101–112 (Sept.).

Callon, R. 1992. "TCP and UDP with Bigger Addresses (TUBA), A Simple Proposal for Internet Addressing and Routing," RFC 1347, 9 pages (June).

Case, J. D., Fedor, M. S., Schoffstall, M. L., and Davin, C. 1990. "Simple Network Management (SNMP)," RFC 1157, 36 pages (May).

The protocol specification for SNMP.

Case, J. D., McCloghrie, K., Rose, M. T., and Waldbusser, S. 1993. "An Introduction to Version 2 of the Internet-Standard Network Management Framework," RFC 1441, 13 pages (Apr.).

An introduction to SNMPv2 along with references to the other 11 RFCs defining SNMPv2.

Case, J. D., and Partridge, C. 1989. "Case Diagrams: A First Step to Diagrammed Management Information Bases," *Computer Communication Review*, vol. 19, no. 1, pp. 13–16 (Jan.).

Defines diagrams that are useful for visualizing the relationship between SNMP variables in a given module.

Casner, S., and Deering, S. E. 1992. "First IETF Internet Audiocast," *Computer Communication Review*, vol. 22, no. 3, pp. 92–97 (July).

Describes how live audio for an IETF meeting was audiocast using multicasting over the Internet. A PostScript copy of this paper is available via anonymous FTP from the host `venera.isi.edu` in the file `pub/ietf-audiocast-article.ps`. Also, the file `mbone/faq.txt` on that host contains the frequently asked questions regarding the Internet multicast backbone (MBONE).

Cheriton, D. P. 1988. "VMTP: Versatile Message Transaction Protocol," RFC 1045, 123 pages (Feb.).

Cheswick, W. R., and Bellovin, S. M. 1994. *Firewalls and Internet Security: Repelling the Wily Hacker.* Addison-Wesley, Reading, Mass.

Describes how to set up and administer a firewall gateway and the security issues involved.

Clark, D. D. 1982. "Window and Acknowledgment Strategy in TCP," RFC 813, 22 pages (July).

The original RFC that identified the silly window syndrome and how to avoid it.

Clark, D. D. 1988. "The Design Philosophy of the DARPA Internet Protocols," *Computer Communication Review*, vol. 18, no. 4, pp. 106–114 (Aug.).

Describes the early reasoning that shaped the Internet protocols.

Comer, D. E., and Stevens, D. L. 1993. *Internetworking with TCP/IP: Vol. III: Client–Server Programming and Applications, BSD Socket Version.* Prentice-Hall, Englewood Cliffs, N.J.

Cooper, A. W., and Postel, J. B. 1993. "The US Domain," RFC 1480, 47 pages (June).

Describes the `.us` domain in the DNS.

Crocker, D. H. 1982. "Standard for the Format of ARPA Internet Text Messages," RFC 822, 47 pages (Aug.).

Defines the format of electronic mail messages transmitted using SMTP.

Crocker, D. H. 1993. "Evolving the System," in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 41–76. Addison-Wesley, Reading, Mass.

Some history on the development of standards in the ARPANET, along with details on the current structure of the Internet technical community. Also defines the current Internet standards process.

Croft, W., and Gilmore, J. 1985. "Bootstrap Protocol (BOOTP)," RFC 951, 12 pages (Sept.).

Crowcroft, J., Wakeman, I., Wang, Z., and Sirovica, D. 1992. "Is Layering Harmful?," *IEEE Network*, vol. 6, no. 1, pp. 20–24 (Jan.).

The seven missing figures from this paper appear in the next issue, vol. 6, no. 2 (March).

Curry, D. A. 1992. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, Reading, Mass.

A book on Unix security. Chapters 4 and 5 deal with network security.

Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., and Lumley, J. 1993. "Afterburner," *IEEE Network*, vol. 7, no. 4, pp. 36–43 (July).

Describes how to speed up TCP by reducing the number of data copies performed, and a special purpose interface card that supports this design.

Danzig, P. B., Obraczka, K., and Kumar, A. 1992. "An Analysis of Wide-Area Name Server Traffic," *Computer Communication Review*, vol. 22, no. 4, pp. 281–292 (Oct.).

An analysis of the traffic to one of the root name servers over two 24-hour periods. Shows how DNS traffic from faulty implementations can consume 20 times more WAN bandwidth than necessary. A PostScript copy of this paper is available via anonymous FTP from the host `caldera.usc.edu` in the file `pub/danzig/dns.ps.Z`.

Deering, S. E. 1989. "Host Extensions for IP Multicasting," RFC 1112, 17 pages (Aug.).

The specification of IP multicasting and IGMP.

Deering, S. E., ed. 1991. "ICMP Router Discovery Messages," RFC 1256, 19 pages (Sept.).

Deering, S. E., and Cheriton, D. P. 1990. "Multicast Routing in Datagram Internetworks and Extended LANs," *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 85–110 (May).

Proposes extensions to common routing techniques to support multicasting.

Dixon, T. 1993. "Comparison of Proposals for Next Version of IP," RFC 1454, 15 pages (May).

A comparison and summary of SIP, PIP, and TUBA.

Droms, R. 1993. "Dynamic Host Configuration Protocol," RFC 1541, 39 pages (Oct.).

Droms, R., and Dyksen, W. R. 1990. "Performance Measurements of the X Window System Communication Protocol," *Software Practice & Experience*, vol. 20, pp. 119–136 (Oct.).

Measurements of the TCP communication involved when using various X clients.

Fedor, M. S. 1988. "GATED: A Multi-routing Protocol Daemon for UNIX," *Proceedings of the 1988 Summer USENIX Conference*, pp. 365–376, San Francisco, Calif.

Finlayson, R. 1984. "Bootstrap Loading using TFTP," RFC 906, 4 pages (June).

Finlayson, R., Mann, T., Mogul, J. C., and Theimer, M. 1984. "A Reverse Address Resolution Protocol," RFC 903, 4 pages (June).

Ford, P. S., Rekhter, Y., and Braun, H-W. 1993. "Improving the Routing and Addressing of IP," *IEEE Network*, vol. 7, no. 3, pp. 10–15 (May).

A description of CIDR (classless interdomain routing).

Fuller, V., Li, T., Yu, J. Y., and Varadhan, K. 1993. "Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy," RFC 1519, 24 pages (Sept.).

The specification of CIDR (classless interdomain routing).

Gerich, E. 1993. "Guidelines for Management of IP Address Space," RFC 1466, 10 pages (May).

The specification of how IP addresses will be allocated in the future (i.e., class B addresses will be hard to obtain and normally a block of class C addresses will be allocated instead).

Gurwitz, R., and Hinden, R. 1982. "IP—Local Area Network Addressing Issues," IEN 212, 11 pages (Sept.).

One of the earliest references to IP broadcast addresses.

Harrenstein, K., Stahl, M. K., and Feinler, E. J. 1985. "NICNAME/WHOIS," RFC 954, 4 pages (Oct.).

Hedrick, C. L. 1988a. "Routing Information Protocol," RFC 1058, 33 pages (June).

Hedrick, C. L. 1988b. "Telnet Terminal Speed Option," RFC 1079, 3 pages (Dec.).

Hedrick, C. L., and Borman, D. A. 1992. "Telnet Remote Flow Control Option," RFC 1372, 6 pages (Oct.).

Hornig, C. 1984. "Standard for the Transmission of IP Datagrams over Ethernet Networks," RFC 894, 3 pages (Apr.).

Huitema, C. 1993. "IAB Recommendation for an Intermediate Strategy to Address the Issue of Scaling," RFC 1481, 2 pages (July).

The IAB recommendation for the implementation of CIDR.

Jacobson, V. 1988. "Congestion Avoidance and Control," *Computer Communication Review*, vol. 18, no. 4, pp. 314–329 (Aug.).

A classic paper describing the slow start and congestion avoidance algorithms for TCP. A PostScript copy of this paper is available via anonymous FTP from the host `ftp.ee.lbl.gov` in the file `congavoid.ps.Z`.

Jacobson, V. 1990a. "Compressing TCP/IP Headers for Low-Speed Serial Links," RFC 1144, 43 pages (Feb.).

Describes CSLIP, a version of SLIP with the TCP and IP headers compressed.

Jacobson, V. 1990b. "Modified TCP Congestion Avoidance Algorithm," April 30, 1990, end2end-interest mailing list (Apr.).

Describes the fast retransmit and fast recovery algorithms.

Jacobson, V. 1990c. "Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno," *Proceedings of the Eighteenth Internet Engineering Task Force*, p. 365 (Sept.), University of British Columbia, Vancouver, B.C.

Jacobson, V., and Braden, R. T. 1988. "TCP Extensions for Long-Delay Paths," RFC 1072, 16 pages (Oct.).

Describes the selective acknowledgment option for TCP, which was removed from the later RFC 1323.

Jacobson, V., Braden, R. T., and Borman, D. A. 1992. "TCP Extensions for High Performance," RFC 1323, 37 pages (May).

Describes the window scale option, the timestamp option, and the PAWS algorithm, along with the reasons these modifications are needed.

Jacobson, V., Braden, R. T., and Zhang, L. 1990. "TCP Extensions for High-Speed Paths," RFC 1185, 21 pages (Oct.).

Despite this RFC being obsoleted by RFC 1323, the appendix on protection against old duplicate segments in TCP is worth reading.

Juszczak, C. 1989. "Improving the Performance and Correctness of an NFS Server," *Proceedings of the 1989 Winter USENIX Conference*, pp. 53–63, San Diego, Calif.

Provides implementation details on an NFS server cache.

Kantor, B. 1991. "BSD Rlogin," RFC 1282, 5 pages (Dec.).

The specification of the Rlogin protocol.

Karn, P., and Partridge, C. 1987. "Improving Round-Trip Time Estimates in Reliable Transport Protocols," *Computer Communication Review*, vol. 17, no. 5, pp. 2–7 (Aug.).

Details of Karn's algorithm to handle the retransmission timeout for segments that have been retransmitted. A PostScript copy of this paper is available via anonymous FTP from the host `sics.se` in the file `pub/craig/karn-partridge.ps`.

Katz, D. 1990. "Proposed Standard for the Transmission of IP Datagrams Over FDDI Networks," RFC 1188, 11 pages (Oct.).

Specifies the encapsulation of IP datagrams and ARP requests and replies on FDDI networks, including multicasting.

Kent, C. A., and Mogul, J. C. 1987. "Fragmentation Considered Harmful," *Computer Communication Review*, vol. 17, no. 5, pp. 390–401 (Aug.).

Kent, S. T. 1991. "U.S. Department of Defense Security Options for the Internet Protocol," RFC 1108, 17 pages (Nov.).

Kleinrock, L. 1992. "The Latency/Bandwidth Tradeoff in Gigabit Networks," *IEEE Communications Magazine*, vol. 30, no. 4, pp. 36–40 (Apr.).

Klensin, J., Freed, N., and Moore, K. 1993. "SMTP Service Extension for Message Size Declaration," RFC 1427, 8 pages (Feb.).

Klensin, J., Freed, N., Rose, M. T., Stefferud, E. A., and Crocker, D. 1993a. "SMTP Service Extensions," RFC 1425, 10 pages (Feb.).

Klensin, J., Freed, N., Rose, M. T., Stefferud, E. A., and Crocker, D. 1993b. "SMTP Service Extension for 8bit-MIME Transport," RFC 1426, 6 pages (Feb.).

Krol, E. 1992. *The Whole Internet*. O'Reilly & Associates, Sebastopol, Calif.

A beginner's introduction to the Internet.

LaQuey, T. 1993. *The Internet Companion: A Beginner's Guide to Global Networking*. Addison-Wesley, Reading, Mass.

A beginner's introduction to the Internet.

Leffler, S. J., and Karels, M. J. 1984. "Trailer Encapsulations," RFC 893, 3 pages (Apr.).

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Mass.

An entire book on the 4.3BSD Unix system. This book describes the Tahoe release of 4.3BSD.

Lougheed, K., and Rekhter, Y. 1991. "A Border Gateway Protocol 3 (BGP-3)," RFC 1267, 35 pages (Oct.).

Lynch, D. C. 1993. "Historical Perspective," in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 3–14. Addison-Wesley, Reading, Mass.

A description of the early days of the Internet: the ARPANET.

Macklem, R. 1991. "Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol," *Proceedings of the 1991 Winter USENIX Conference*, pp. 53–64, Dallas, Tex.

Describes an implementation of NFS that uses both UDP and TCP.

Malkin, G. S. 1993a. "RIP Version 2: Carrying Additional Information," RFC 1388, 7 pages (Jan.).

Malkin, G. S. 1993b. "Traceroute Using an IP Option," RFC 1393, 7 pages (Jan.).

Proposed modifications to ICMP for a new version of traceroute.

Mallory, T., and Kullberg, A. 1990. "Incremental Updating of the Internet Checksum," RFC 1141, 2 pages (Jan.).

Describes an implementation technique for incrementally updating the internet checksum.

Manber, U. 1990. "Chain Reactions in Networks," *IEEE Computer*, vol. 23, no. 10, pp. 57–63 (Oct.).

Describes types of broadcast storms and network meltdowns, similar to those demonstrated in Exercises 9.3 and 9.4.

McCanne, S., and Jacobson, V. 1993. "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," *Proceedings of the 1993 Winter USENIX Conference*, pp. 259–269, San Diego, Calif.

A detailed description of the BSD Packet Filter (BPF) and comparisons with Sun's Network Interface Tap (NIT).

McCloghrie, K., and Rose, M. T. 1991. "Management Information Base for Network Management of TCP/IP-based Internets: MIB-II," RFC 1213 (Mar.).

McGregor, G. 1992. "PPP Internet Protocol Control Protocol (IPCP)," RFC 1332, 12 pages (May).

A description of the NCP for PPP that is specific to TCP/IP.

Mills, D. L. 1992. "Network Time Protocol (Version 3): Specification, Implementation, and Analysis," RFC 1305, 113 pages (Mar.).

Mockapetris, P. V. 1987a. "Domain Names: Concepts and Facilities," RFC 1034, 55 pages (Nov.).

An introduction to the DNS.

Mockapetris, P. V. 1987b. "Domain Names: Implementation and Specification," RFC 1035, 55 pages (Nov.).

The specification of the DNS.

Mogul, J. C. 1990. "Efficient Use of Workstations for Passive Monitoring of Local Area Networks," *Computer Communication Review*, vol. 20, no. 4, pp. 253–263 (Sept.).

Describes the use of workstations to monitor local area networks, instead of purchasing dedicated network analyzer hardware.

Mogul, J. C. 1992. "Holy Turbocharger Batman, (evil cheating), NFS async writes," Message-ID <1992Mar2.191711.9935@PA.dec.com>, Usenet, comp.protocols.nfs Newsgroup (Mar.).

Some interesting statistics on internet checksum errors collected on a busy NFS server over 40 days.

Mogul, J. C. 1993. "IP Network Performance," in *Internet System Handbook*, eds. D. C. Lynch and M. T. Rose, pp. 575–675. Addison-Wesley, Reading, Mass.

Covers numerous topics in the TCP/IP protocols that are candidates for tuning to obtain optimal performance.

Mogul, J. C., and Deering, S. E. 1990. "Path MTU Discovery," RFC 1191, 19 pages (Apr.).

Mogul, J. C., and Postel, J. B. 1985. "Internet Standard Subnetting Procedure," RFC 950, 18 pages (Aug.).

Moore, K. 1993. "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text," RFC 1522, 10 pages (Sept.).

Describes a way to send non-ASCII characters in RFC 822 mail headers, using 7-bit ASCII.

Moy, J. 1991. "OSPF Version 2," RFC 1247, 189 pages (July).

Nagle, J. 1984. "Congestion Control in IP/TCP Internetworks," RFC 896, 9 pages (Jan.).

Description of the Nagle algorithm.

Nye, A., ed. 1992. *The X Window System, Volume 0: X Protocol Reference Manual, Third Edition.* O'Reilly & Associates, Sebastopol, Calif.

Obraczka, K., Danzig, P. B., and Li, S. 1993. "Internet Resource Discovery Services," *IEEE Computer*, vol. 26, no. 9, pp. 8–22 (Sept.).

Presents an overview of current Internet resource discovery tools: Alex, Archie, Gopher, Indie, Knowbot Information Service, Netfind, Prospero, WAIS, WWW, and X.500. A PostScript copy of this paper is available via anonymous FTP from the host `caldera.usc.edu` in the file `/pub/kobraczk/ieeecomputer.ps.Z`.

Papadopoulos, C., and Parulkar, G. M. 1993. "Experimental Evaluation of SunOS IPC and TCP/IP Protocol Implementation," *IEEE/ACM Transactions on Networking*, vol. 1, no. 2, pp. 199–216 (Apr.).

Measures the overhead involved at different layers of the protocol suite as data is sent and received.

Partridge, C. 1986. "Mail Routing and the Domain System," RFC 974, 7 pages (Jan.).

How to use DNS MX records for mail routing.

Partridge, C. 1994. *Gigabit Networking*. Addison-Wesley, Reading, Mass.

Describes the issues involved when network s peeds exceed one gigabit per second.

Partridge, C., and Pink, S. 1993. "A Faster UDP," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 429–440 (Aug.).

Describes implementation improvements to the Berkeley sources to speed up UDP performance about 30%.

Paxson, V. 1993. "Empirically-Derived Analytic Models of Wide-Area TCP Connections: Extended Report," LBL-34086, Lawrence Berkeley Laboratory and EECS Division, University of California, Berkeley (June).

Contains an analysis of 2.5 million TCP connections that occurred during 14 wide-area traffic traces. A PostScript copy of this report is available via anonymous FTP from the host `ftp.ee.lbl.gov` in the files `WAN-TCP-models.1.ps.Z` and `WAN-TCP-models.2.ps.Z`.

Perlman, R. 1992. *Interconnections: Bridges and Routers*. Addison-Wesley, Reading, Mass.

A book with lots of details on ways to interconnect networks (bridges and routers) along with various routing algorithms.

Plummer, D. C. 1982. "An Ethernet Address Resolution Protocol," RFC 826, 10 pages (Nov.).

Postel, J. B. 1980. "User Datagram Protocol," RFC 768, 3 pages (Aug.).

Postel, J. B., ed. 1981a. "Internet Protocol," RFC 791, 45 pages (Sept.).

Postel, J. B. 1981b. "Internet Control Message Protocol," RFC 792, 21 pages (Sept.).

Postel, J. B., ed. 1981c. "Transmission Control Protocol," RFC 793, 85 pages (Sept.).

Postel, J. B. 1982. "Simple Mail Transfer Protocol," RFC 821, 68 pages (Aug.).

Postel, J. B. 1987. "TCP and IP Bake Off," RFC 1025, 6 pages (Sept.).

Describes some of the procedures and scoring performed between different implementations of TCP/IP during its early development phases to test for interoperability.

Postel, J. B., ed. 1994. "Internet Official Protocol Standards," RFC 1600, 36 pages (Mar.).

The status of all Internet protocols. This RFC is updated regularly—check the latest RFC index for the current version.

Postel, J. B., and Reynolds, J. K. 1983a. "Telnet Protocol Specification," RFC 854, 15 pages (May).

The basic Telnet protocol specification. Many later RFCs describe specific Telnet options.

Postel, J. B., and Reynolds, J. K. 1983b. "Telnet Binary Transmission," RFC 856, 4 pages (May).

Postel, J. B., and Reynolds, J. K. 1983c. "Telnet Echo Option," RFC 857, 5 pages (May).

Postel, J. B., and Reynolds, J. K. 1983d. "Telnet Suppress Go Ahead Option," RFC 858, 3 pages (May).

Postel, J. B., and Reynolds, J. K. 1983e. "Telnet Status Option," RFC 859, 3 pages (May).

Postel, J. B., and Reynolds, J. K. 1983f. "Telnet Timing Mark Option," RFC 860, 4 pages (May).

Postel, J. B., and Reynolds, J. K. 1985. "File Transfer Protocol (FTP)," RFC 959, 69 pages (Oct.).

Postel, J. B., and Reynolds, J. K. 1988. "Standard for the Transmission of IP Datagrams over IEEE 802 Networks," RFC 1042, 15 pages (Apr.).

The specification for the encapsulation of IP datagrams and ARP requests and replies on IEEE 802 networks.

Pusateri, T. 1993. "IP Multicast Over Token-Ring Local Area Networks," RFC 1469, 4 pages (June).

Rago, S. A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, Mass.

A book on TLI and the streams subsystem.

Rekhter, Y., and Gross, P. 1991. "Application of the Border Gateway Protocol in the Internet," RFC 1268, 13 pages (Oct.).

Rekhter, Y., and Li, T. 1993. "An Architecture for IP Address Allocation with CIDR," RFC 1518, 27 pages (Sept.).

Reynolds, J. K. 1989. "The Helminthiasis of the Internet," RFC 1135, 33 pages (Dec.).

Contains a detailed discussion of the Internet worm of 1988.

Reynolds, J. K., and Postel, J. B. 1992. "Assigned Numbers," RFC 1340, 138 pages (July).

All the magic numbers in the Internet protocol suite. This RFC is update regularly—check the latest RFC index for the current version.

Romkey, J. L. 1988. "A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP," RFC 1055, 6 pages (June).

Rose, M. T. 1990. *The Open Book: A Practical Perspective on OSI*. Prentice-Hall, Englewood Cliffs, N.J.

A book on the OSI protocols. Chapter 8 provides additional details on ASN.1 and BER.

Rose, M. T. 1993. *The Internet Message: Closing the Book with Electronic Mail.* Prentice-Hall, Englewood Cliffs, N.J.

A book on Internet mail, with details on MIME.

Rose, M. T. 1994. *The Simple Book: An Introduction to Internet Management, Second Edition.* Prentice-Hall, Englewood Cliffs, N.J.

A book on SNMPv2. The first edition of this book covered SNMPv1.

Rose, M. T., and McCloghrie, K. 1990. "Structure and Identification of Management Information for TCP/IP-based Internets," RFC 1155, 22 pages (May).

Defines the SMI for SNMPv1.

Rosenberg, W., Kenney, D., and Fisher, G. 1992. *Understanding DCE.* O'Reilly & Associates, Sebastopol, Calif.

Provides an overview of OSF's Distributed Computing Environment.

Routhier, S. A. 1993. "Implementation Experience for SNMPv2," *The Simple Times*, vol. 2, no. 4, pp. 1–4 (July-Aug.).

Describes modifying an SNMPv1 implementation to support SNMPv2.

This journal is distributed electronically at no charge. Send e-mail with a subject of `help` to `st-subscriptions@simple-times.org` to receive subscription information.

Schryver, V. J. 1993. "Info on High Speed Transport Protocols Requested," Message-ID <i0imr8g@rhyolite.wpd.sgi.com>, Usenet, comp.protocols.tcp-ip Newsgroup (May).

Provides TCP performance numbers for some FDDI implementations.

Schwartz, M. F., and Tsirigotis, P. G. 1991. "Experience with a Semantically Cognizant Internet White Pages Directory Tool," *Journal of Internetworking Research and Experience*, vol. 2, no. 1, pp. 23–50 (Mar.).

Also available by anonymous FTP from the host `ftp.cs.colorado.edu` in the file `pub/cs/techreports/schwartz/PostScript/White.Pages.ps.Z`.

Simpson, W. A. 1993. "The Point-to-Point Protocol (PPP)," RFC 1548, 53 pages (Dec.).

Defines PPP and its link control protocol.

Sollins, K. R. 1992. "The TFTP Protocol (Revision 2)," RFC 1350, 11 pages (July).

Stallings, W. 1987. *Handbook of Computer-Communications Standards, Volume 2: Local Network Standards.* Macmillan, New York.

Contains details on the IEEE 802 local area network standards.

Stallings, W. 1993. *SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards.* Addison-Wesley, Reading, Mass.

Describes the differences between SNMPv1 and SNMPv2.

Stern, H. 1991. *Managing NFS and NIS.* O'Reilly & Associates, Sebastopol, Calif.

Contains lots of details on installing, using, and administering NFS.

Stevens, W. R. 1990. *UNIX Network Programming.* Prentice-Hall, Englewood Cliffs, N.J.

A detailed book on network programming under Unix, using sockets and TLI.

Stevens, W. R. 1992. *Advanced Programming in the UNIX Environment.* Addison-Wesley, Reading, Mass.

A detailed book on Unix programming.

Sun Microsystems. 1987. "XDR: External Data Representation Standard," RFC 1014, 20 pages (June).

Sun Microsystems. 1988a. "RPC: Remote Procedure Call, Protocol Specification, Version 2," RFC 1057, 25 pages (June).

Sun Microsystems. 1988b. "NFS: Network File System Protocol Specification," RFC 1094, 27 pages (Mar.).

The specification of version 2 of Sun NFS.

Sun Microsystems. 1994. *NFS: Network File System Version 3 Protocol Specification*. Sun Microsystems, Mountain View, Calif.

A PostScript copy of this document is available by anonymous FTP from the host `ftp.uu.net` in the file `networking/ip/nfs/NFS3.spec.ps.Z`.

Tanenbaum, A. S. 1989. *Computer Networks, Second Edition*. Prentice-Hall, Englewood Cliffs, N.J.

A general text on computer networks.

Topolcic, C. 1993. "Status of CIDR Deployment in the Internet," RFC 1467, 9 pages (Aug.).

Tsuchiya, P. F. 1991. "On the Assignment of Subnet Numbers," RFC 1219, 13 pages (Apr.).

Recommends assigning subnet IDs from the highest order bit down, and host IDs from the lowest order bit up. This makes it easier to change the subnet mask at some point in the future without having to renumber all the systems.

Ullmann, R. 1993. "TP/IX: The Next Internet," RFC 1475, 35 pages (June).

Another proposal for the next generation of Internet protocols.

VanBokkelen, J. 1989. "Telnet Terminal-Type Option," RFC 1091, 7 pages (Feb.).

Waitzman, D. 1988. "Telnet Window Size Option," RFC 1073, 4 pages (Oct.).

Waitzman, D., Partridge, C., and Deering, S. E. 1988. "Distance Vector Multicast Routing Protocol," RFC 1075, 24 pages (Nov.).

Warnock, R. P. 1991. "Need Help Selecting Ethernet Cards for Very High Performance Throughput Rates," Message-ID <lbhal10@sgi.sgi.com>, Usenet, comp.protocols.tcp-ip Newsgroup (Sept.).

Provides the TCP performance numbers we calculated from Figure 24.9.

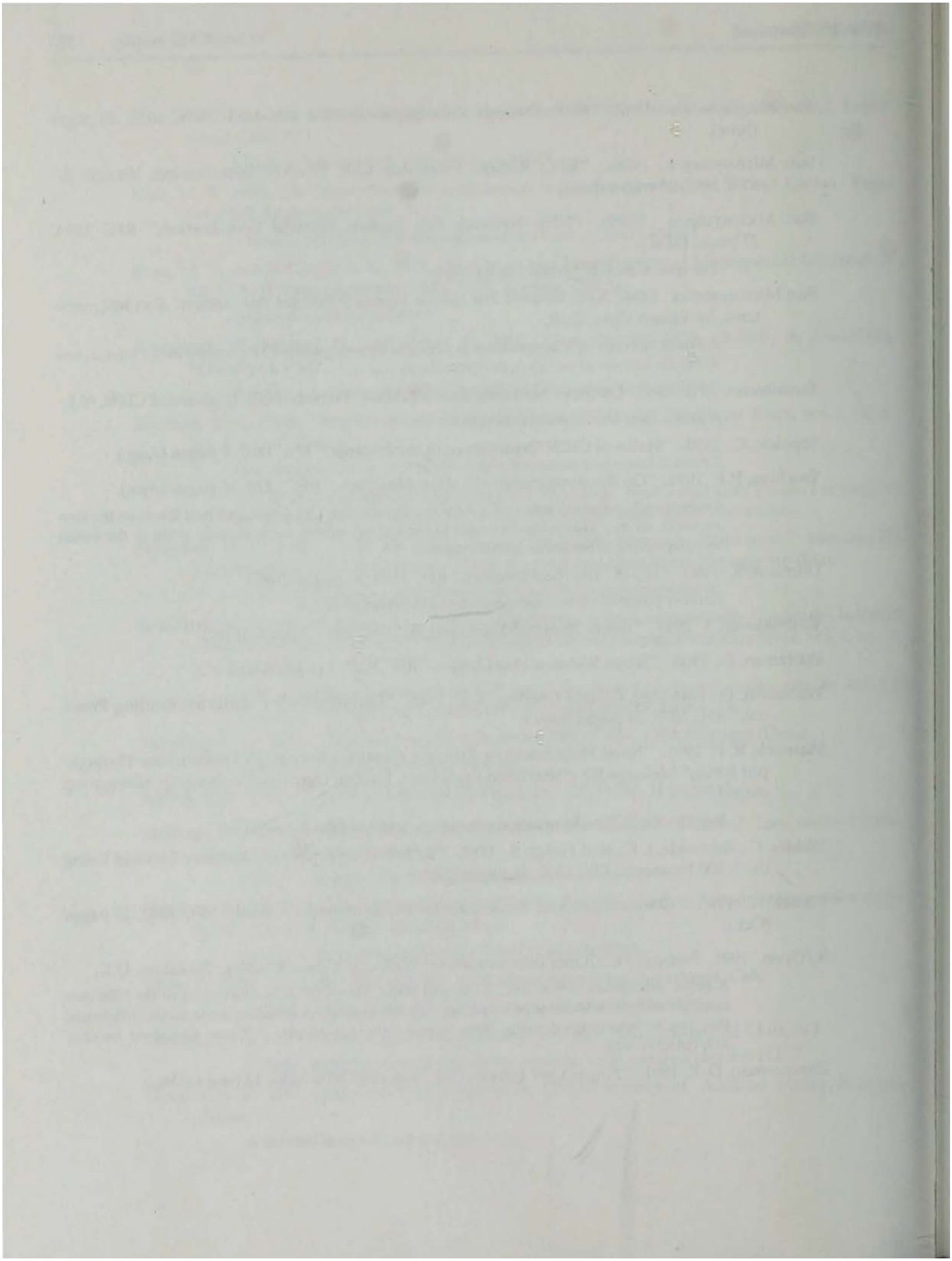
Weider, C., Reynolds, J. K., and Heker, S. 1992. "Technical Overview of Directory Services Using the X.500 Protocol," RFC 1309, 16 pages (Mar.).

Wimer, W. 1993. "Clarifications and Extensions for the Bootstrap Protocol," RFC 1542, 23 pages (Oct.).

X/Open. 1991. *Protocols for X/Open Internetworking: XNFS*. X/Open, Reading, Berkshire, U.K.

A better description of Sun RPC, XDR, and NFS. Also contains a description of the NFS lock manager and status monitor protocols, along with appendices detailing the semantic differences that can be encountered using NFS, versus local file access. X/Open document number XO/CAE/91/030.

Zimmerman, D. P. 1991. "Finger User Information Protocol," RFC 1288, 12 pages (Dec.).



Index

Rather than provide a separate glossary (with most of the entries being acronyms), this index also serves as a glossary for all the acronyms used in the book. The primary entry for the acronym appears under the acronym name. For example, all references to the Network Time Protocol appear under NTP. The entry under the compound term "Network Time Protocol" refers the reader back to the main entry under NTP. This is because you are more likely to look up the acronym than the compound term. Additionally, a list of all these acronyms with their compound terms is found in the inside back cover.

2MSL
  timer, TCP, 297, 345
  wait, TCP, 242–246, 261, 352, 357, 429, 431, 505, 517–518, 522, 534
4.2BSD, xviii, 17, 25, 119, 126, 172, 282, 335, 391, 409, 512, 528
4.3BSD, xviii, 17, 126, 172, 282, 335, 548
  Reno, xviii, 17, 35, 160, 311
  Tahoe, 17, 311, 548
4.4BSD, xviii, 17, 29, 48, 63, 81–82, 116, 118, 123, 163, 167, 232, 251, 253, 282, 345, 347, 349, 392, 406, 411, 445, 491, 496
  configuration options, 537–538
8-bit encoding, SMTP, 456

A record, DNS, 193, 201, 204–205, 446
abortive release, 247, 505
absolute domain name, DNS, 189
Abstract Syntax Notation One, *see* ASN.1
Accetta, M., 492
ACK (acknowledgment flag, TCP header), 227
  compression, 519
  number, TCP, 226
  smashing, 519
  TCP delayed, 265–267, 277, 312, 350, 353, 418, 528, 534
acknowledgment flag, TCP header, *see* ACK
active
  close, 233
  open, 232
Adams, R., 25
address
  all-subnets-directed broadcast, 45, 172
  broadcast, 73
  class A IP, 8
  class B IP, 8
  class C IP, 8
  class D IP, 8, 175, 177–178, 180

class E IP, 8
hardware, 22, 57
IP, 7–9
limited broadcast, 45, 136, 171, 173, 217–218
multicast group, 175–176
net-directed broadcast, 45, 171
protocol, 57
special case IP, 45
subnet, 42–45, 550, 553
subnet-directed broadcast, 45, 47, 136, 172–174, 178
address mask request–reply, ICMP, 72–73, 222, 532, 538
picture of, ICMP, 72
Address Resolution Protocol, *see* ARP
Adelman, K., 97
Advanced Research Projects Agency network, *see* ARPANET
agent, SNMP, 359
AIX, xviii, 81, 116, 123, 159, 163, 237, 244, 282, 406, 433, 491
3.2.2 configuration options, 536–537
Albitz, P., 197, 543
Alex, 550
Alexander, S., 221, 543
alias, DNS, 201
aliasing, 349
all-hosts group address, 183
all-subnets-directed broadcast address, 45, 172
Almquist, P., 15, 35, 46, 83, 98, 128, 151, 161, 172, 543–544
American Standard Code for Information Interchange, *see* ASCII
anonymous, FTP, 436–438, 539
ANY record, DNS, 193
API (application program interface), 17–18
AppleTalk, 26, 374–375
application
gateway, 4
layer, 3
program interface, *see* API
Archie, 485, 539, 550
ARP (Address Resolution Protocol), 53–64, 550
and RARP, request–reply, picture of, 56
cache, 56, 374, 379, 533
example, 54–60
flooding, 158
gratuitous, 62–63, 543
hack, 62
interaction, UDP and, 157–159
nonexistent host, 59–60
packet format, 56–57
promiscuous, 62
proxy, 60–62, 120–121
ARPANET (Advanced Research Projects Agency network), 15, 91, 98, 138, 389, 545, 548
`arp` program, 56, 60–61, 63–64, 126, 220, 508, 530
`-a` option, 56, 63–64
`-d` option, 63
`-s` option, 63, 220
`arpa` domain, DNS, 189, 198
`arp_cache_report` variable, 533
`arp_cleanup_interval` variable, 533
`arp_debug` variable, 533
`arpt_killc` variable, 536
AS (autonomous system), 128
multihomed, 139
stub, 139
transit, 139
ASCII (American Standard Code for Information Interchange)
NVT, 363, 401–402, 421, 423, 426, 432, 442, 445–447, 453–457, 481–483, 504, 516
ASN.1 (Abstract Syntax Notation One), 386–387, 551
assassination, TCP TIME_WAIT, 518, 544
Assigned Numbers RFC, 14, 19, 98, 176, 365, 372, 403–404
asynchronous
link, PPP, 26
WRITE, NFS, 479
`at` group, SNMP, 374–375
audiocast, 545
authoritative answer, DNS, 192, 196, 199
autonomous system, *see* AS
AXFR record, DNS, 193

backoff, exponential, 81, 299, 301, 305, 318, 325, 519, 521
bake-off, TCP/IP, 230, 551
bandwidth-delay product, TCP, 289–291, 317, 344–345, 356, 519
Banks, D., 356, 546
base-64 encoding, SMTP, 455–456
Basic Encoding Rules, *see* BER
Bellovin, S. M., xix, 342, 344, 483, 543, 545
BER (Basic Encoding Rules), 386–387, 551
Berkeley Internet Name Domain, *see* BIND
Berkeley Packet Filter, *see* BPF
Berkeley Software Distribution, *see* BSD
Berkeley-derived implementations, 16
BGP (Border Gateway Protocol), 129, 138–139, 548, 551
keepalive message, 139
Bhide, A., 63, 543
bibliography, alphabetical, 543–553
big packets or small packets?, TCP, 342–344
big-endian byte order, 34

binary
encoding, SMTP, 456
transmission option, Telnet, 403, 551
BIND (Berkeley Internet Name Domain), 188, 196, 515, 523, 541, 543
body, *see* SMTP body
BOOTP (Bootstrap Protocol), 215–222, 543, 545, 553
chicken and egg, 220
example, 218–219
message, picture of, 216
packet format, 215–217
server, 541
server design, 219–220
through a router, 220
transaction ID, 217–218, 222
vendor-specific information, 221–222
Bootstrap Protocol, *see* BOOTP
Border Gateway Protocol, *see* BGP
Borenstein, N., 456, 543
Borman, D. A., xix, 15, 37, 253, 339, 355, 405, 408, 544, 547
Bostic, K., xix
BPF (Berkeley Packet Filter), 68, 491–493, 542, 549
Braden, R. T., 15, 19, 37, 46, 83, 98, 161, 245, 253, 339, 345, 352, 518, 543–544, 547
Braun, H-W., 119, 140–141, 546
bridge, 6, 514, 550
broadcast, 8, 54, 171–172, 505
examples, 172–175
IP address, 169, 546
storm, 71, 549
broadcast address, 73
all-subnets-directed, 45, 172
limited, 45, 136, 171, 173, 217–218
net-directed, 45, 171
subnet-directed, 45, 47, 136, 172–174, 178
BSD (Berkeley Software Distribution), 16
Networking Release 1.0, 17
Networking Release 2.0, xvii, 17, 540
Networking Release 3.0, 17
packet filter, 491–493
BSD/386, xvii, 48, 73, 87–88, 93, 116, 119, 159, 163, 236–237, 244, 264, 282, 406, 408, 411, 413, 432–433, 449, 491, 495, 500–501, 519
configuration options, 526–527
`bufmod` streams module, 494
bulk data
flow, TCP, 275–296
throughput, TCP, 286–292
byte order, 465, 489
big-endian, 34
little-endian, 34
network, 34, 395
byte stream service, 224
Caceres, R., 263, 441, 544
cache
ARP, 56, 374, 379, 533
DNS, 194, 203–206, 208
NFS server, 479
Calamvokis, C., 356, 546
Callon, R., 50, 544
carrier sense multiple access, *see* CSMA
Case diagrams, 367, 545
Case, J. D., 360, 367, 387, 545
Casnet, S., 186, 545
`cat` program, 398–399, 474, 476
character-at-a-time mode, Telnet, 404, 406–411
`chargen` server, 16
checksum
algorithm, 36, 544, 549
errors, 147, 168, 513, 549
Ethernet, 147, 170
ICMP, 69, 228, 380, 531
IGMP, 180, 228, 531
IP, 36, 147, 170, 228, 343, 376, 513, 516, 530–531, 536
link layer, 22, 24
modem, 90
OSPF, 142, 513
TCP, 147, 224, 227–228, 356, 381, 531
UDP, 144–147, 170, 211, 228, 356, 366, 513, 516, 527, 529, 531, 533, 538
Cheriton, D. P., 186, 354, 545–546
Cheswick, W. R., 483, 545
chicken and egg, BOOTP, 220
Christmas tree packet, 230
CIDR (classless interdomain routing), 50, 140–141, 546–547, 551, 553
CIX (Commercial Internet Exchange), 119
Clark, D. D., 51, 325, 545
Clark, J. J., xix
class
A IP address, 8
B IP address, 8
C IP address, 8
D IP address, 8, 175, 177–178, 180
E IP address, 8
classless interdomain routing, *see* CIDR
cleartext password, 137–138, 362, 388, 392, 411, 417, 427, 496
client, 3
escapes, Rlogin, 395–396
escapes, Telnet, 406
interrupt, Rlogin, 393
stub, RPC, 462
to server commands, Rlogin, 395
client–server model, 12
CLNP (Connectionless Network Protocol), 50, 141

clock synchronization, 74–77, 349, 351
clocks, computer, 499–501
close
  active, 233
  passive, 233
  TCP half-duplex, 262
close function, 238
CLOSED state, TCP, 240–241, 246, 261, 381, 517
CLOSE_WAIT state, TCP, 240–241, 246, 261, 363, 381
CLOSING state, TCP, 241, 252, 381
CNAME record, DNS, 193, 201, 205–206, 446, 449–450
Comer, D. E., 391, 545
Commercial Internet Exchange, *see* CIX
community, SNMP, 362, 370
compress program, 437
compressed SLIP, *see* CSLIP
computer clocks, 499–501
concurrent server, 12, 213, 254, 257–258, 488, 505
config program, 526–527
configuration options, 525–538
  4.4BSD, 537–538
  AIX 3.2.2, 536–537
  BSD/386, 526–527
  Solaris 2.2, 529–536
  SunOS 4.1.3, 527–529
  SVR4, 529
congestion
  avoidance algorithm, TCP, 310–312, 547
  example, TCP, 306–309, 313–316
  TCP, 291–292
  window, 285
connection
  establishment, TCP, 229–234
  establishment timer, TCP, 235–236, 534
  incarnation, 243
  management, FTP, 425–431
  termination, TCP, 229–234
connectionless, 33, 229
Connectionless Network Protocol, *see* CLNP
connection-oriented, 223, 229
content, *see* SMTP content
control connection, FTP, 419
convergence, routing, 138
cookie, magic, 221
Cooper, A. W., 189, 545
Coordinated Universal Time, *see* UTC
Counter datatype, SNMP, 363
country domains, DNS, 189
CRC (cyclic redundancy check), 22, 27
credentials, RPC, 464
cretin, 48
Crocker, D. H., 14, 442, 452–453, 545, 548
Croft, W., 215, 545
Crowcroft, J., xix, 518, 545
CSLIP (compressed SLIP), 25, 27, 31–32, 48, 96, 274, 510, 518, 540, 547
  encapsulation, 25
CSMA (carrier sense multiple access), 21, 176
Curry, D. A., 392, 481, 546
cwnd variable, 285, 310–317, 534
cyclic redundancy check, *see* CRC

daemon, 111
  routing, 127
  Unix routing, 128–129
Dalton, C., 356, 546
Danzig, P. B., 188, 263, 441, 485, 544, 546, 550
data connection, FTP, 420
Data Link Provider Interface, *see* DLPI
data mark command, Telnet, 406
data representation, FTP, 421–422
datagram, 2, 7
  size, UDP maximum, 159–160
  truncation, UDP, 160
data-link layer, 2
date program, 60, 264
Davin, C., 360, 545
daytime server, 16, 411
DCE (Distributed Computing Environment), 77, 462, 552
DDN (Defense Data Network), 8
debug option, socket, 301, 303–304, 313, 496–497, 505
DECnet, 26
Deering, S. E., 29–30, 97, 123, 153, 179, 185–186, 340, 545–546, 549, 553
Defense Data Network, *see* DDN
delay
  propagation, 289
  transmission, 289
delayed ACK, TCP, 265–267, 277, 312, 350, 353, 418, 528, 534
demultiplexing, 11, 36, 168, 213, 255, 262, 431, 518
design
  BOOTP server, 219–220
  RARP server, 67–68
  TCP server, 254–260
  TFTP server, 212
  UDP server, 162–167
DeSimone, A., xix
Destination Service Access Point, *see* DSAP
/dev/arp device, 530
/dev/icmp device, 530
/dev/ip device, 530
/dev/nit device, 493
/dev/tcp device, 530
/dev/udp device, 530

DF (don't fragment flag, IP header), 149, 151–153, 155, 340, 532
DHCP (Dynamic Host Configuration Protocol), 222, 543, 546
dig program, 197, 541
dinosaur, 421, 461
direct route, 113
directed broadcasts, forwarding, 174–175, 526, 528, 531
DIRECTED_BROADCAST constant, 526, 528
discard server, 16, 411
DISPLAY environment variable, 408, 489
DisplayString datatype, SNMP, 363
Distance–Vector Multicast Routing Protocol, *see* DVMRP
distance-vector routing protocol, 137
Distributed Computing Environment, *see* DCE
Distributed Time Service, *see* DTS
Dixon, T., 50, 546
DLPI (Data Link Provider Interface), 68, 494–495
DNS (Domain Name System), 9, 187–208
  A record, 193, 201, 204–205, 446
  absolute domain name, 189
  alias, 201
  ANY record, 193
  arpa domain, 189, 198
  authoritative answer, 192, 196, 199
  AXFR record, 193
  cache, 194, 203–206, 208
  CNAME record, 193, 201, 205–206, 446, 449–450
  country domains, 189
  example, 194–198, 206–207
  generic domains, 189
  geographical domains, 189
  HINFO record, 193, 201
  hostname spoofing, 200
  iterative query, 192, 204–205
  label, 192
  message, 191–194
  message, picture of, 191
  MX record, 193, 201–202, 448–451, 550
  MX record preference value, 202, 448–451, 459
  NS record, 193, 202, 204, 515
  organizational domains, 189
  pointer, 197
  pointer query, 198–200
  primary name server, 190
  PTR record, 193, 199–201, 438
  query class, 193
  recursive query, 192, 196, 204–205, 207
  root name server, 190–191, 204–205, 208, 515, 546
  RR, 194, 201–202
  RR, picture of, 193
  secondary name server, 190
  timeout and retransmission, 206
  truncated response, 192, 206
  TTL, 194, 202
  UDP or TCP, 206
  zone, 190
  zone transfer, 190
doc program, 197, 541
dogfight, 176
domain directive, 195, 203
domain literal, 459, 523
domain name, DNS absolute, 189
Domain Name System, *see* DNS
don't fragment flag, IP header, *see* DF
dotted-decimal notation, 7
Droms, R., 221–222, 488, 543, 546
DSAP (Destination Service Access Point), 22
ds.internic.net, 8, 484–485
DTS (Distributed Time Service), 77
DVMRP (Distance–Vector Multicast Routing Protocol), 185, 553
Dyksen, W. R., 488, 546
Dynamic Host Configuration Protocol, *see* DHCP
dynamic routing, 127–142

EBONE (European IP Backbone), 119
echo option, Telnet, 403, 551
echo request–reply, ICMP, 85–96, 175, 493, 510–512, 532
  picture of, ICMP, 86
echo server, 16, 411
Edwards, A., 356, 546
EGP (Exterior Gateway Protocol), 128–129
Einstein, A., 347
Elnozahy, E. N., 63, 543
emacs program, 393
encapsulation, 9–10
  CSLIP, 25
  Ethernet, 21–23, 168, 237, 547
  FDDI, 548
  IEEE 802, 21–23, 168, 237, 513, 551
  PPP, 26–27
  SLIP, 24–25
  trailer, 23–24, 548
encoding
  SMTP 8-bit, 456
  SMTP base-64, 455–456
  SMTP binary, 456
  SMTP quoted-printable, 454–456
encrypted password, 496
end of option list, *see* EOL
end system, 4
end-to-end protocol, 4
enterprise, SNMP, 386

envelope, *see* SMTP envelope
environment
  option, Telnet, 403, 408, 544
  variable, `DISPLAY`, 408, 489
  variable, `PRINTER`, 408
EOL (end of option list), 93, 108
ephemeral port number, 13, 533, 535
ESTABLISHED state, TCP, 240–241, 250, 252, 254–255, 330, 363, 381, 517, 521
`/etc/conf/pack.d/ip/space.c` file, 529
`/etc/defaultrouter` file, 116
`/etc/ethers` file, 67
`/etc/hosts` file, 54
`/etc/inet/rc.inet` file, 116
`/etc/netstart` file, 116
`/etc/networks` file, 113
`/etc/passwd` file, 496
`/etc/rc2.d/S69inet` file, 116
`/etc/rc.local` file, 116
`/etc/rc.net` file, 116
`/etc/resolv.conf` file, 195, 200
`/etc/services` file, 13
`/etc/shadow` file, 496
`etherfind` program, 495
Ethernet, 21
  checksum, 147, 170
  encapsulation, 21–23, 168, 237, 547
  frame, picture of, 23
  frame type field, 22, 57–58, 65, 68
  meltdown, 512
  MTU, 30
  multicasting, 176–177
European IP Backbone, *see* EBONE
exercises, solutions to, 507–524
expanding ring search, 182
EXPN command, SMTP, 451–452
exponential backoff, 81, 299, 301, 305, 318, 325, 519, 521
extended SMTP, SMTP futures, 452–454
Exterior Gateway Protocol, *see* EGP
external data representation, *see* XDR

fast retransmit and fast recovery algorithm, TCP, 309, 312–313, 344, 527, 534, 547
FCS (frame check sequence), 22
FDDI (Fiber Distributed Data Interface), 4, 178, 345, 355, 521, 548
  encapsulation, 548
  MTU, 30
  multicasting, 178, 548
Fedor, M. S., 128, 360, 545–546
Feinler, E. J., 483, 547
Fiber Distributed Data Interface, *see* FDDI
FIFO (first in, first out), 164, 259
file
  format control, FTP, 421
  handle, NFS, 469
  structure, FTP, 421
  transmission mode, FTP, 422
  type, FTP, 421
File Transfer Protocol, *see* FTP
FIN (finish flag, TCP header), 227
Finger protocol, 481–483, 553
finish flag, TCP header, *see* FIN
Finlayson, R., 65, 172, 546
FIN_WAIT_1 state, TCP, 241, 252, 330, 381
FIN_WAIT_2 state, TCP, 241, 246, 261, 330, 381
firewall gateway, 85, 201, 483, 545
first in, first out, *see* FIFO
Fisher, G., 77, 552
flooding, ARP, 158
Florence, R., 449
flow control
  Rlogin, 393
  TCP, 224, 227, 280–282, 310
  Telnet, 408
Ford, P. S., 119, 140–141, 546
`fork` function, 254, 505
`.forward` file, 445
forwarding
  directed broadcasts, 174–175, 526, 528, 531
  IP, 119, 526, 528–529, 531, 536, 538
FQDN (fully qualified domain name), 189, 197, 199, 207, 372
fragmentation, 29, 548
  IP, 148–150, 168
  offset field, IP, 149, 158
fragmentation required, ICMP, 79, 149, 151–157, 340, 534
  picture of, ICMP, 151
frame, 9
  check sequence, *see* FCS
  type field, Ethernet, 22, 57–58, 65, 68
Freed, N., 452–453, 456, 543, 548
FTP (File Transfer Protocol), 419–439
  aborting file transfer, 433–436
  anonymous, 436–438, 539
  commands, 423
  connection management, 425–431
  control connection, 419
  data connection, 420
  data representation, 421–422
  examples, 426–438
  file format control, 421
  file structure, 421
  file transmission mode, 422
  file type, 421
  protocol, 419–426, 551
  replies, 423–425

ftp program, -d option, 426–427
full-duplex, TCP, 15, 226, 233, 239, 459
Fuller, V., 140, 546
fully qualified domain name, *see* FQDN
futures
  extended SMTP, SMTP, 452–454
  IP, 49–50
  MIME, SMTP, 456–458, 542
  non-ASCII headers, SMTP, 454–455
  SMTP, 452–458
  TCP, 339–354

garbage byte, 335
gated program, 111, 128, 132, 201, 540, 546
gateway, 4
  application, 4
GATEWAY constant, 526
gateway, firewall, 85, 201, 483, 545
Gateway Requirements RFC, 15, 544
Gauge datatype, SNMP, 363
generic domains, DNS, 189
geographical domains, DNS, 189
Gerich, E., 140, 546
get-bulk-request operator, SNMP, 387
gethostbyaddr function, 187, 199–200, 207
gethostbyname function, 54, 187, 198
get-next-request operator, SNMP, 360
get-request operator, SNMP, 360, 370–371
get-response operator, SNMP, 360
gettimeofday function, 499–500
gigabit network, 345–347, 548, 550
Gilligan, R., xix
Gilmore, J., 215, 545
Gitlin, J., xix
Godsil, J. M., xix
Gopher, 485, 550
Grandi, S., xix
gratuitous ARP, 62–63, 543
Gross, P., 139, 551
group address
  all-hosts, 183
  multicast, 175–176
Gulbenkian, J., xix
Gurwitz, R., 172, 546

hack, 35
  ARP, 62
half-close, TCP, 233, 238–240, 504
half-duplex
  close, TCP, 262
  mode, Telnet, 404
half-open, TCP, 248
handshake, TCP three-way, 231, 352
hard mount, NFS, 478
hardware address, 22, 57
Harrenstein, K., 483, 547
Haverlock, P. M., xix
HDLC (high-level data link control), 26
header, *see* SMTP header
  length field, IP, 34, 36–37, 91, 144
  length field, TCP, 226, 228
Hedrick, C. L., 129, 132, 408, 547
Heker, S., 484, 553
HELLO (routing protocol), 128–129
helminthiasis, 551
Herbert, T., xix
high-level data link control, *see* HDLC
Hinden, R., 172, 546
HINFO record, DNS, 193, 201
HIPPI, 355
Hogue, J. E., xix
hop-by-hop protocol, 4
Hornig, C., 22, 172, 547
host group, 176
  permanent, 176
host program, 196, 199, 203–206, 446, 524, 541
  -t option, 201
  -v option, 202, 204
Host Requirements RFC, 15, 19, 22, 60, 64, 73, 81, 83, 105, 114, 146, 158, 163, 171, 213, 222–223, 232, 262, 267, 270, 280, 293, 331, 333, 347, 416, 422, 429, 431, 433, 436, 448, 508, 515, 535, 544
host unreachable, ICMP, 117–119, 317–319
hostname spoofing, DNS, 200
HP-UX, 491
Huitema, C., 140, 547
Hyperchannel MTU, 30
hypertext, 486

IAB (Internet Architecture Board), 14, 140
IANA (Internet Assigned Number Authority), 13, 16, 176, 396, 452
ICMP (Internet Control Message Protocol), 69–83, 179–186, 550
  4.4BSD processing of, 81–82
  address mask request–reply, 72–73, 222, 532, 538
  address mask request–reply, picture of, 72
  checksum, 69, 228, 380, 531
  echo request–reply, 85–96, 175, 493, 510–512, 532
  echo request–reply, picture of, 86
  errors, TCP, 317–318
  fragmentation required, 79, 149, 151–157, 340, 534
  fragmentation required, picture of, 151
  host unreachable, 117–119, 317–319

message types, 70–71
network unreachable, 117–119, 317, 336
port unreachable, 77–81, 83, 98, 100, 165, 170, 247, 509, 511, 514
redirect, 39, 93, 95, 119–123, 126, 510, 512, 526, 528–529, 531–532, 536, 538
redirect, picture of, 122
router advertisement, picture of, 124
router discovery, 123–125, 510, 540, 546
router solicitation, picture of, 124
RTT measurement, 75
source quench, 160–162, 164, 317, 510, 532
source quench, picture of, 161
time exceeded, 98, 100, 106–107, 158, 182, 384, 510–511
time exceeded, picture of, 100
timestamp request–reply, 74–77, 96, 510, 532
timestamp request–reply, picture of, 74
unreachable, 510
unreachable, picture of, 79
icmp group, SNMP, 379
icmpaddrmask program, 72, 540
icmp_bsd_compat variable, 532
icmp_def_ttl variable, 532
icmptime program, 74, 540
icmp_wroff_extra variable, 533
idempotent procedures, NFS, 478–479
identification field, IP, 36, 149–150, 513
IDRP (Interdomain Routing Protocol), 141
IEEE (Institute of Electrical and Electronics Engineers), 21, 176
802 encapsulation, 21–23, 168, 237, 513, 551
802.3/802.2 MTU, 30
IEN (Internet Experiment Notes), 172
IESG (Internet Engineering Steering Group), 14
IETF (Internet Engineering Task Force), 14, 545
ifconfig program, 29, 33, 47–49, 51, 61, 116–117, 173, 377, 514, 525, 530
-a option, 48
IGMP (Internet Group Management Protocol), 546
checksum, 180, 228, 531
message, 180
message, picture of, 180
protocol, 180–183
reports and queries, 181
TTL, 182
IGP (interior gateway protocol), 128
implementations, Berkeley-derived, 16
TCP/IP, 16–17
in-addr.arpa domain, 198–200
in-band signaling, 395
incarnation, connection, 243
incoming connection request queue, TCP, 257–260, 505, 534
Indie, 550
indirect route, 113
inet_addr function, 173
inform-request operator, SNMP, 387
initial sequence number, *see* ISN
input queue, UDP, 163–164
in.rdisc program, 125
instance identification
simple variables, SNMP, 368
tables, SNMP, 368
Institute of Electrical and Electronics Engineers, *see* IEEE
INTEGER datatype, SNMP, 363
interactive
data flow, TCP, 263–274
input, TCP, 263–265
Interdomain Routing Protocol, *see* IDRP
interface group, SNMP, 372–374
interface, loopback, 28–29, 48, 51, 73, 96, 114–117, 159, 183, 355, 382, 486, 491, 510, 531, 536
interior gateway protocol, *see* IGP
intermediate system, 4
Intermediate System to Intermediate System Protocol, *see* IS–IS
International Organization for Standardization , *see* ISO
internet, 4, 16
layer, 2
Internet, 1, 8, 16, 19, 102, 154, 485–486, 548
worm, 481, 546, 551
Internet Architecture Board, *see* IAB
Internet Assigned Number Authority, *see* IANA
Internet Control Message Protocol, *see* ICMP
Internet Engineering Steering Group, *see* IESG
Internet Engineering Task Force, *see* IETF
Internet Experiment Notes, *see* IEN
Internet Group Management Protocol, *see* IGMP
Internet Official Protocol Standards, 15
Internet Protocol, *see* IP
Internet Research Task Force, *see* IRTF
Internet Society, *see* ISOC
InterNIC, 8, 20, 102, 191, 208, 458, 483–485
ioctl function, 220
IP (Internet Protocol), 33–51, 550
checksum, 36, 147, 170, 228, 343, 376, 513, 516, 530–531, 536
datagram, 9, 150
forwarding, 119, 526, 528–529, 531, 536, 538
fragment offset field, 149, 158
fragment retransmission, 149
fragmentation, 148–150, 168
futures, 49–50
header, 34–37
header length field, 34, 36–37, 91, 144
header, picture of, 34
identification field, 36, 149–150, 513
options, 37, 168