CHAPTER

File Transfer Protocol 20

What You Will Learn

In this chapter, you will learn how FTP provides a method to move files around the Internet. We'll examine various aspects of FTP as a protocol and as an application, showing how commands translate to protocol actions.

You will learn about the differences between FTP's active and passive modes of operation. We'll discuss how security concerns affect the operation of FTP.

The original Internet boasted three applications: electronic mail, remote computer access, and remote file access. Over time, not only have these three been joined by a host of others but the original applications have evolved to keep pace with expansion of the Internet and the environment of the modern world. As a simple example of this trend, these applications have all moved beyond their simple commands typed in at a prompt to graphical front ends. These GUIs make the applications more accessible to novices, but at the same time mask the details of protocol operation from users. Yet in most cases the original protocols are still there, running behind the scenes, as this look at the File Transfer Protocol (FTP) will show.

FTP transfers a copy of a file. The original file is usually still present on the source host, available for copying over and over as remote users request it. Copying files between two different computer systems has always been more difficult than it seems. Today, most users are familiar with the differences between Windows file formats and those used by Apple, which is why one can't usually take a floppy or CD from one and load it on the other. When other file systems are considered, such as the varieties of Unix and older formats used by minicomputer and mainframe vendors (many of which could not be copied between computer models from the same *vendor*), it is no wonder the FTP is one of the most elaborate and robust applications in TCP/IP (although format conversion is much less of a concern than it used to be).

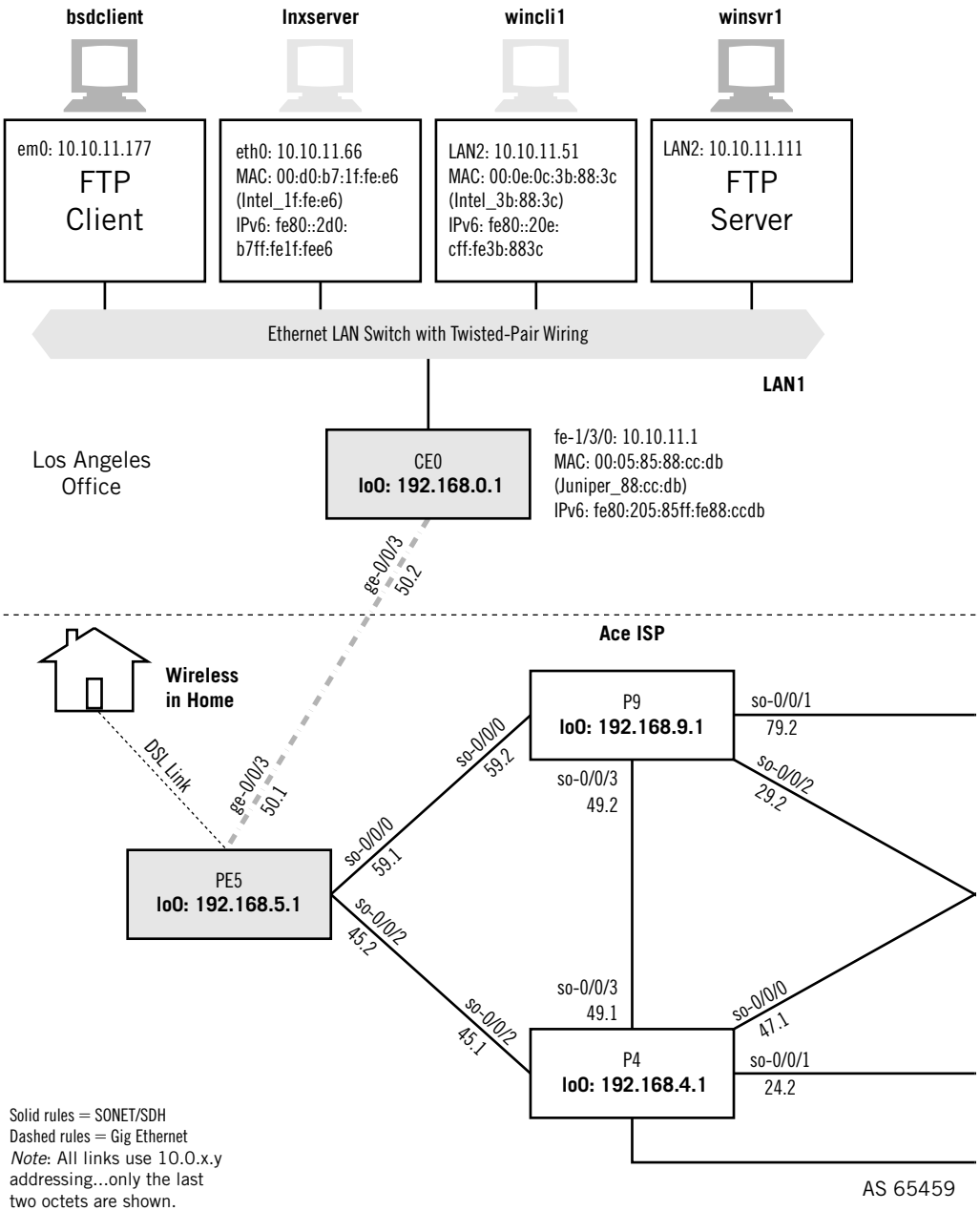
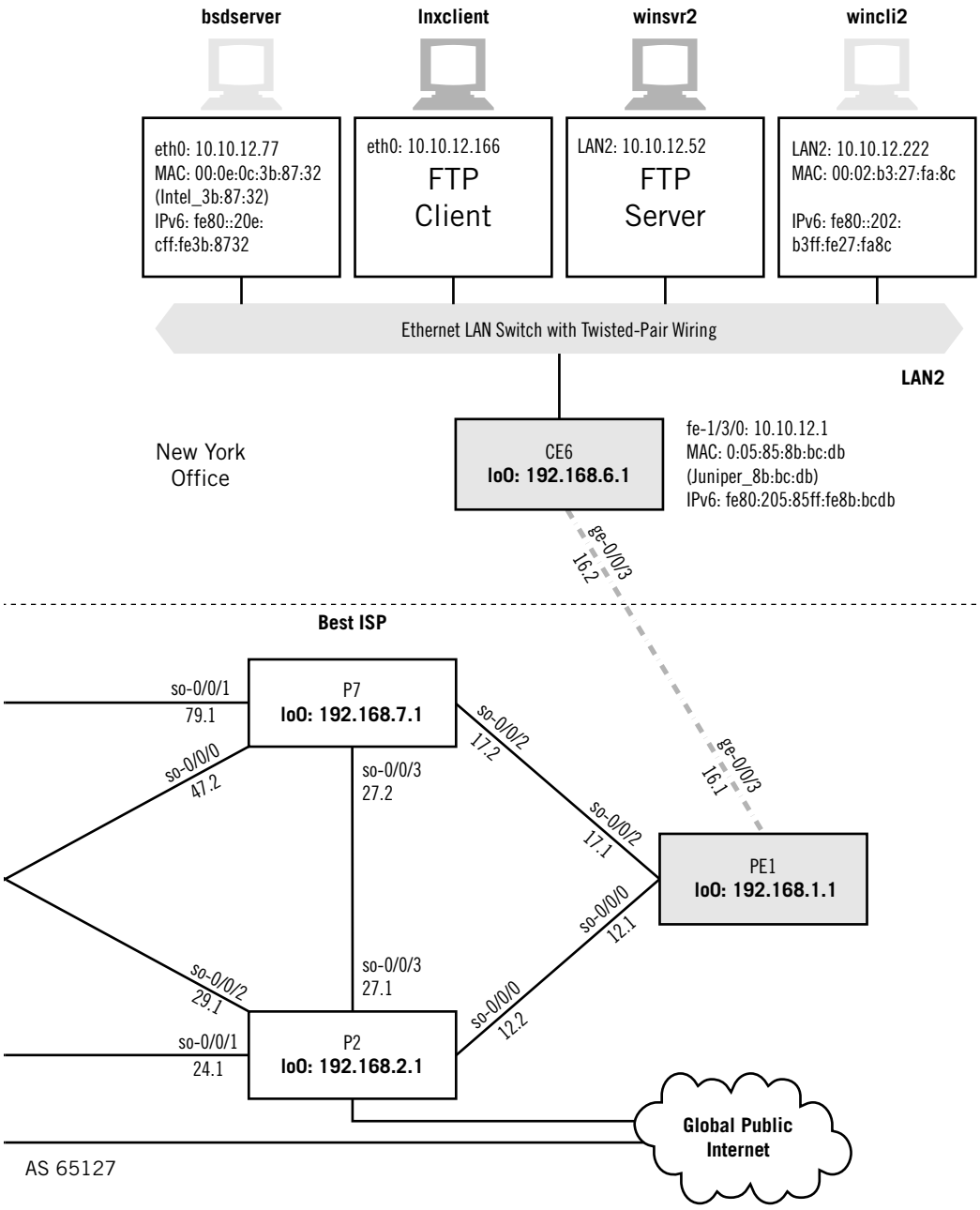


FIGURE 20.1

FTP client and servers on the Illustrated Network use Unix-based and Windows hosts.


bsdserver
lnxclient
winsvr2
wincli2
eth0: 10.10.12.77
MAC: 00:0e:0c:3b:87:32
(Intel_3b:87:32)
IPv6: fe80::20e:
cff:fe3b:8732
eth0: 10.10.12.166
FTP
Client
LAN2: 10.10.12.52
FTP
Server
LAN2: 10.10.12.222
MAC: 00:02:b3:27:fa:8c
IPv6: fe80::202:
b3ff:fe27:fa8c
Ethernet LAN Switch with Twisted-Pair Wiring
LAN2
New York
Office
CE6
lo0: 192.168.6.1
fe-1/3/0: 10.10.12.1
MAC: 0:05:85:8b:bc:db
(Juniper_8b:bc:db)
IPv6: fe80:205:85ff:fe8b:bcdb
ge-0/0/3
16.2
Best ISP
so-0/0/1
79.1
P7
lo0: 192.168.7.1
so-0/0/2
17.2
so-0/0/0
47.2
so-0/0/3
27.2
ge-0/0/3
16.1
so-0/0/2
17.1
PE1
lo0: 192.168.1.1
so-0/0/0
12.1
so-0/0/2
29.1
so-0/0/3
27.1
so-0/0/0
12.2
so-0/0/1
24.1
P2
lo0: 192.168.2.1
Global Public
Internet
AS 65127

OVERVIEW

Of all the applications covered in this book, FTP is the one we've used most on the Illustrated Network. Whenever we had software to install, capture files to consolidate, or screen images to transfer, we used FTP to move them around. Every server device had a different FTP package installed, from the "native" FreeBSD and Linux CLI version to a couple of different GUI FTP servers for Windows XP.

That said, the "experimental" nature of the Illustrated Network should be noted. FTP is still useful for file transfers on the global public Internet (especially a form known as *anonymous FTP*), but in the real world it's better practice to use an authenticated form of file transfer such as SFTP or SCP (discussed at the end of this chapter). Let's take a look at how these applications look and feel. Then we'll explore the basics of FTP operation in a little more detail. This chapter makes FTP servers out of `winsrv1` and `winsrv2`. We'll access them from `bsdclient` and `lnxclient`, as shown in Figure 20.1.

The CLI versions of FTP depend on commands, of course. The GUI version depends on commands as well, but these are often hidden from the user (some show the commands executed after you click on a button or icon). This is not an FTP tutorial, and FTP application's commands are not part of the FTP protocol, but this will give a feel for the number of things FTP can do. You can look at the commands a client can use to tell the servers what to do in FreeBSD and Linux. These are the FTP `help` command listings. The following is FreeBSD:

```
bsdclient# ftp
ftp>> help
Commands may be abbreviated.  Commands are:

!          chmod      ftp      ls       msend     proxy     rhelp     system
$          close      get      macdef   newer     put       rmdir     tenex
account    cr         gate     mdelete  nlist     pwd       rstatus   trace
append     debug      glob     mdir     nmap      quit      runique   type
ascii      delete     hash     mget     ntrans    quote     send      umask
bell       dir        help     mkdir    open      recv      sendport  user
binary     disconnect idle     mls      page      reget     site      verbose
bye        edit       image    mode     passive   rename    size      ?
case       epsv4      lcd      modtime  preserve  reset     status
cd         exit       less     more     progress  restart   struct
cdup       form       lpwd     mput     prompt    restrict  sunique
ftp>>
```

The list given by Linux is similar, but not the same. Most of the commands appear in both lists, but 6 are unique to Linux and 11 are unique to FreeBSD. Some are quite handy, such as the ability in FreeBSD's FTP to `preserve` the modification timestamp on downloaded files. Usually, the "extra" commands are used to determine how files are handled before or after they are transferred. The actual session commands are fairly consistent, and they both get the job done.

The biggest difference in FTP application-level operation is between the "regular" use of the `port` command and the use of the `passive` (PASV) command. Until recently, it was the server that supplied the port number assignment to use for the data connection and then opened the connection. But in *passive mode* the port number and open command used for the data connection is supplied by the client instead of the server, mainly to satisfy firewall rules and still allow FTP to function. We'll talk more about this later in this chapter, because it can cause problems when firewalls are in use, which should be just about always today.

First, let's see if the FreeBSD or Linux versions of Unix differ in how their FTP client implementations handle the PASV mode. In both cases, we'll fetch the same file from the FTP server running on `winsrv1`.

PORT and PASV

In both FreeBSD and Linux, passive mode is the default. The FTP passive command is a toggle that turns the mode on and off as it is entered.

```
ftp> passive
Passive mode off.
ftp> passive
Passive mode on.
ftp>
```

The following shows a little 30,000-byte file called `testfile.zip` from the CLI on FreeBSD and Linux. This example uses a plain text password, but only for instructional purposes.

```
bsdclient# ftp
ftp> open 10.10.11.111
Connected to 10.10.11.111.
220 Fastream NETFile FTP Server Ready
Name (10.10.11.111:admin): walter
331 Password required for walter.
Password: (not shown)
230 User walter logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get testfile.zip
local: testfile.zip remote: testfile.zip
227 Entering Passive Mode (10,10,11,111,7,69).
150 Opening data connection for testfile.zip.
100%
|*****************************************************************************
**************************| 30642       00:00 ETA
226 File sent ok
30642 bytes received in 0.10 seconds (306.08 KB/s)
ftp>
```

FTP Features

Most features that you get by default in some FTP applications (such as the transfer progress "tick marks") must be explicitly turned on in other FTP implementations.

We like the fact that the client shows we are in passive mode and tells me the port number I'm going to use to open the data connection to the server. We also like the tick mark progress bar and the statistics displayed. Let's look at what we get in Linux:

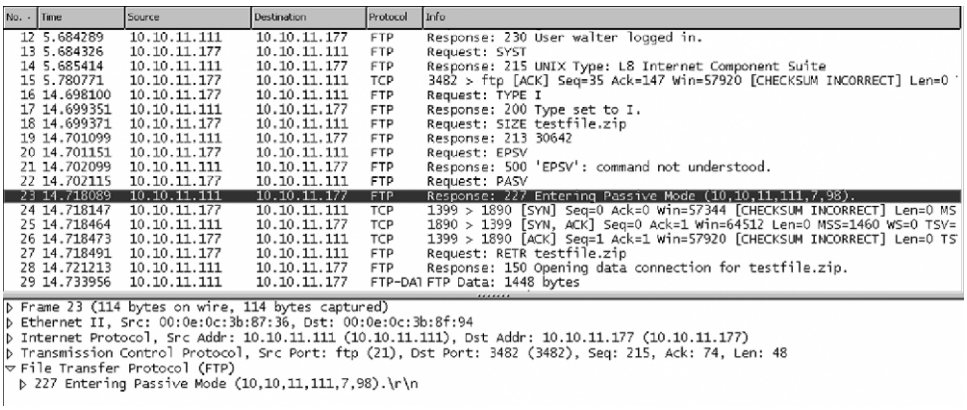
```
[root @lnxclient admin]# ftp
ftp> open 10.10.11.111
Connected to 10.10.11.111.
220 Fastream NETFile FTP Server Ready
500 'AUTH': command not understood.
500 'AUTH': command not understood.
KERBEROS_V4 rejected as an authentication type
Name (10.10.11.111:admin): walter
331 Password required for walter.
Password: (not shown)
230 User walter logged in.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> get testfile.zip
local: testfile.zip remote: testfile.zip
227 Entering Passive Mode (10,10,11,111,7,80).
150 Opening data connection for testfile.zip.
226 File sent ok
30642 bytes received in 0.0065 seconds (4.6e+03 Kbytes/s)
ftp>
```

Linux is more terse and tries to use Kerberos (a more secure authentication method), going back to simple userID and password only when it has to. We are comparing variants of the default FTP client on these systems rather than something built into the systems themselves or a high-quality FTP application. However, we'll look at the packet capture as well.

Let's see what these exchanges look like when captured by Ethereal. Figure 20.2 shows the packets from the time the user logs into the server until that data connection is used.

It is reassuring to note that the client does indeed use the port expected by the server ($7 \times 256 = 1792 + 98 = 1890$), although the port is not in the currently accepted range for these ports. Figure 20.3 shows the same using the Linux client.

As expected with an application as widely used and as venerable as FTP, there are only a few differences here and there. Note that the Windows XP file server identifies

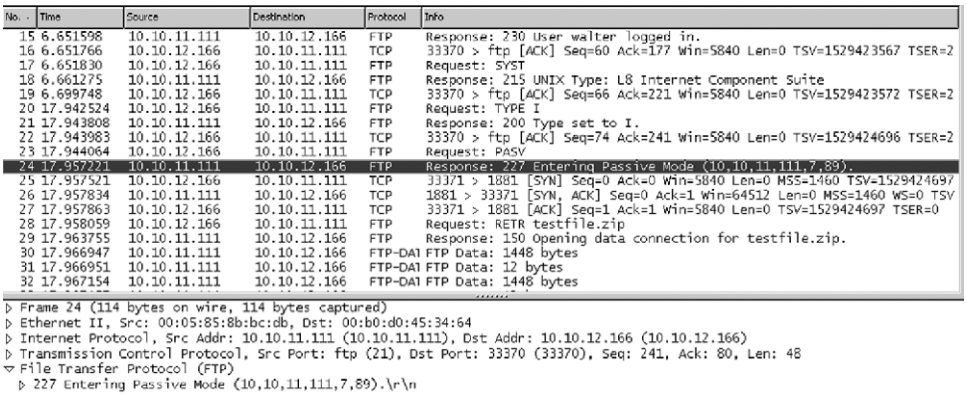


No.	Time	Source	Destination	Protocol	Info
12	5.684289	10.10.11.111	10.10.11.177	FTP	Response: 230 User walter logged in.
13	5.684326	10.10.11.177	10.10.11.111	FTP	Request: SYST
14	5.685414	10.10.11.111	10.10.11.177	FTP	Response: 215 UNIX Type: L8 Internet Component Suite
15	5.780771	10.10.11.177	10.10.11.111	TCP	3482 > ftp [ACK] Seq=35 Ack=147 Win=57920 [CHECKSUM INCORRECT] Len=0
16	14.698100	10.10.11.177	10.10.11.111	FTP	Request: TYPE I
17	14.699351	10.10.11.111	10.10.11.177	FTP	Response: 200 Type set to I.
18	14.699371	10.10.11.177	10.10.11.111	FTP	Request: SIZE testfile.zip
19	14.701099	10.10.11.111	10.10.11.177	FTP	Response: 213 30642
20	14.701151	10.10.11.177	10.10.11.111	FTP	Request: EPSV
21	14.702099	10.10.11.111	10.10.11.177	FTP	Response: 500 'EPSV': command not understood.
22	14.702115	10.10.11.177	10.10.11.111	FTP	Request: PASV
23	14.718089	10.10.11.111	10.10.11.177	FTP	Response: 227 Entering Passive Mode (10,10,11,111,7,98).
24	14.718147	10.10.11.177	10.10.11.111	TCP	1399 > 1890 [SYN] Seq=0 Ack=0 Win=57344 [CHECKSUM INCORRECT] Len=0 MS
25	14.718464	10.10.11.111	10.10.11.177	TCP	1890 > 1399 [SYN, ACK] Seq=0 Ack=1 Win=64512 Len=0 MSS=1460 WS=0 TSV=
26	14.718473	10.10.11.177	10.10.11.111	TCP	1399 > 1890 [ACK] Seq=1 Ack=1 Win=57920 [CHECKSUM INCORRECT] Len=0 TS
27	14.718491	10.10.11.177	10.10.11.111	FTP	Request: RETR testfile.zip
28	14.721213	10.10.11.111	10.10.11.177	FTP	Response: 150 Opening data connection for testfile.zip.
29	14.733956	10.10.11.111	10.10.11.177	FTP-DA	FTP Data: 1448 bytes

```
▷ Frame 23 (114 bytes on wire, 114 bytes captured)
▷ Ethernet II, Src: 00:0e:0c:3b:87:36, Dst: 00:0e:0c:3b:8f:94
▷ Internet Protocol, Src Addr: 10.10.11.111 (10.10.11.111), Dst Addr: 10.10.11.177 (10.10.11.177)
▷ Transmission Control Protocol, Src Port: ftp (21), Dst Port: 3482 (3482), Seq: 215, Ack: 74, Len: 48
▽ File Transfer Protocol (FTP)
  ▷ 227 Entering Passive Mode (10,10,11,111,7,98).\r\n
```

FIGURE 20.2

FTP passive using FreeBSD, showing that the client initiates the data connection.



No.	Time	Source	Destination	Protocol	Info
15	6.651598	10.10.11.111	10.10.12.166	FTP	Response: 230 User walter logged in.
16	6.651766	10.10.12.166	10.10.11.111	TCP	33370 > ftp [ACK] Seq=60 Ack=177 Win=5840 Len=0 TSV=1529423567 TSER=2
17	6.651830	10.10.12.166	10.10.11.111	FTP	Request: SYST
18	6.661275	10.10.11.111	10.10.12.166	FTP	Response: 215 UNIX Type: L8 Internet Component Suite
19	6.699748	10.10.12.166	10.10.11.111	TCP	33370 > ftp [ACK] Seq=66 Ack=221 Win=5840 Len=0 TSV=1529423572 TSER=2
20	17.942524	10.10.12.166	10.10.11.111	FTP	Request: TYPE I
21	17.943808	10.10.11.111	10.10.12.166	FTP	Response: 200 Type set to I.
22	17.943983	10.10.12.166	10.10.11.111	TCP	33370 > ftp [ACK] Seq=74 Ack=241 Win=5840 Len=0 TSV=1529424696 TSER=2
23	17.944064	10.10.12.166	10.10.11.111	FTP	Request: PASV
24	17.957221	10.10.11.111	10.10.12.166	FTP	Response: 227 Entering Passive Mode (10,10,11,111,7,89).
25	17.957521	10.10.12.166	10.10.11.111	TCP	33371 > 1881 [SYN] Seq=0 Ack=0 Win=5840 Len=0 MSS=1460 TSV=1529424697
26	17.957834	10.10.11.111	10.10.12.166	TCP	1881 > 33371 [SYN, ACK] Seq=0 Ack=1 Win=64512 Len=0 MSS=1460 WS=0 TSV
27	17.957863	10.10.12.166	10.10.11.111	TCP	33371 > 1881 [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=1529424697 TSER=0
28	17.958059	10.10.12.166	10.10.11.111	FTP	Request: RETR testfile.zip
29	17.963755	10.10.11.111	10.10.12.166	FTP	Response: 150 Opening data connection for testfile.zip.
30	17.966947	10.10.11.111	10.10.12.166	FTP-DA	FTP Data: 1448 bytes
31	17.966951	10.10.11.111	10.10.12.166	FTP-DA	FTP Data: 12 bytes
32	17.967154	10.10.11.111	10.10.12.166	FTP-DA	FTP Data: 1448 bytes

```
▷ Frame 24 (114 bytes on wire, 114 bytes captured)
▷ Ethernet II, Src: 00:05:85:8b:bc:db, Dst: 00:b0:d0:45:34:64
▷ Internet Protocol, Src Addr: 10.10.11.111 (10.10.11.111), Dst Addr: 10.10.12.166 (10.10.12.166)
▷ Transmission Control Protocol, Src Port: ftp (21), Dst Port: 33370 (33370), Seq: 241, Ack: 80, Len: 48
▽ File Transfer Protocol (FTP)
  ▷ 227 Entering Passive Mode (10,10,11,111,7,89).\r\n
```

FIGURE 20.3

FTP passive using Linux. The port numbers are more in line with current practice.

itself as a "Unix Type" file server. FreeBSD tries an initial `EPSV`, the RFC 2428 extended passive command for IPv6, and network address translation (NAT) environments and FTP. (We'll talk all about `EPSV` later in the chapter.) It then uses, as Linux does from the start, the `PASV` command.

Linux is more in line with current client port usage conventions, using 33371 rather than FreeBSD, which still is using four-digit port numbers. In both cases, the data transfer does not use the well-known port 20 on the server side.

FTP AND GUIS

When it comes to Windows, `winsrv1` is running the FTP package Fastream and `winsrv2` is running FileZilla. We had no familiarity with these packages: they were just the first "shareware" ones we found when looking on the Web. Again, given the history of vulnerabilities in FTP servers and the possible consequences of having a server subverted you should not run random FTP software found on the Internet except in tightly controlled circumstances like these.

The Fastream NETFile FTP server is also an HTTP Web server and is free for personal use. It has a nice logging capability, which can display on-screen and save to a file at the same time. This is shown in Figure 20.4.

FileZilla has the most impressive array of log-in variations, as shown in Figure 20.5. We'll say more about SSL and SSH in later chapters. SFTP solves many of the problems running FTP with tunnels and NAT can cause.

In addition, almost all Web browsers can handle FTP as well as HTTP, the Web protocol. This is part of the "universal client" role of the browser.

For example, if we use the Web browser on `winsrv1` to "visit" the FTP server on `winsrv2` (`ftp://winsrv2`), we are still asked to log in (no anonymous user is defined on `winsrv2`, but if it had been, no log-in screen would appear. The log-in request is shown in Figure 20.6.

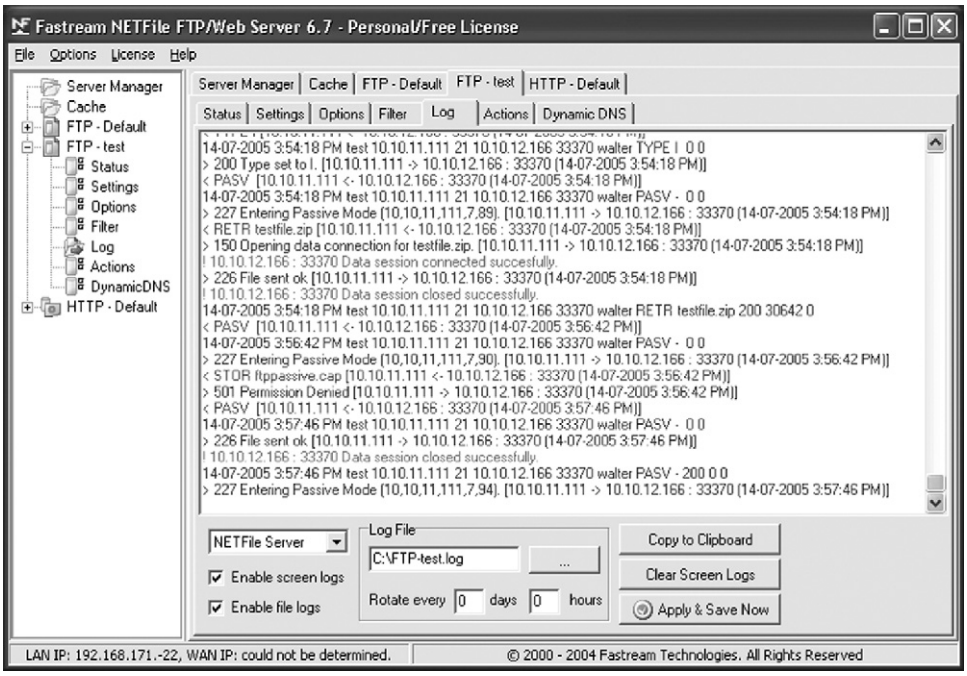


FIGURE 20.4

Fastream FTP logging. Note the amount of detail provided.

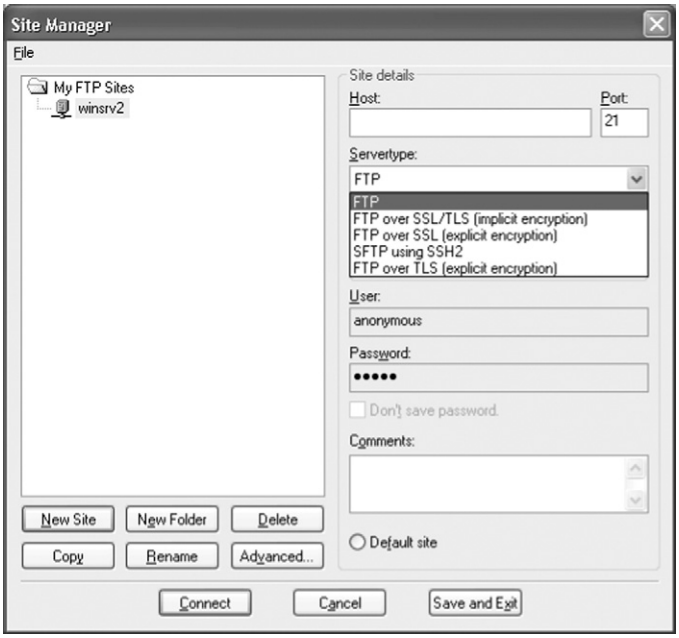


FIGURE 20.5

FileZilla FTP log-in variations. SFTP is part of SSH2, but is a separate protocol.

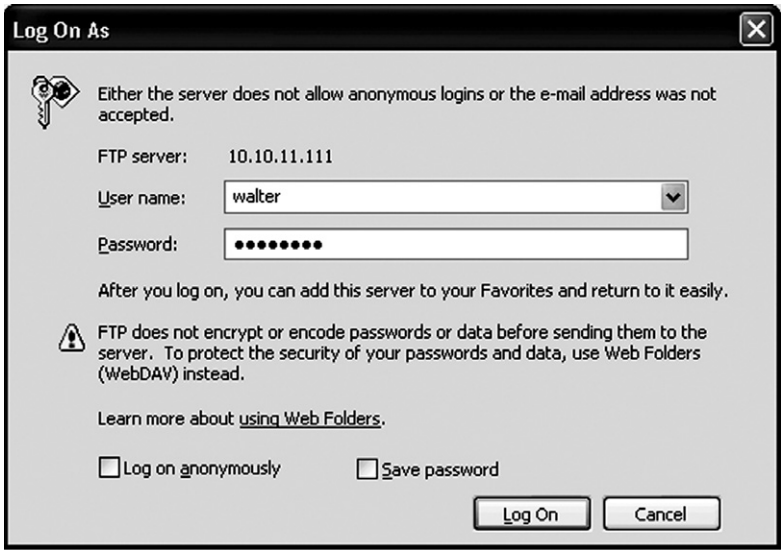


FIGURE 20.6

FTP browser log-in screen, showing how verbose a GUI can be compared to CLI implementations.

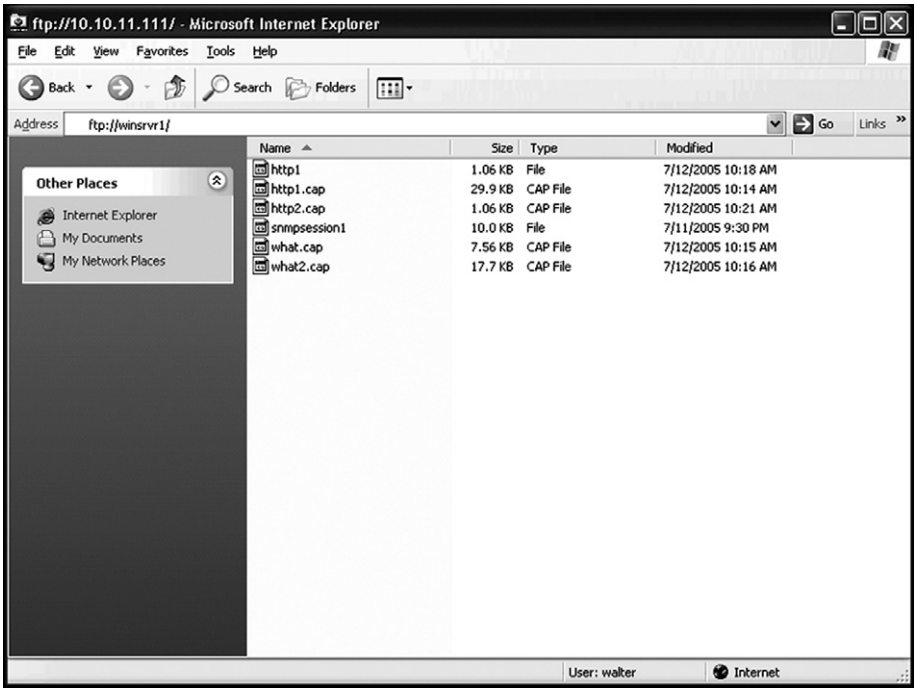


FIGURE 20.7

Browser FTP listing, showing how a browser can act as a "universal client."

But once we log in properly, we will get a listing of the default FTP directory. This directly, `C:\NFRoot`, contained a series of Ethereal capture files when this was done (as shown in Figure 20.7).

FTP Basics

FTP was defined in RFC 959 and updated in RFC 2228, RFC 2640, RFC 2773, RFC 3659 and several others. One major difference between FTP and almost every other application is the fact that FTP employs not one but *two* ports between client and server. One explanation is that there is always an available control connection to quickly countermand actions that have unintended or unexpected results. But RFC 959 simply notes that the control connection essentially uses the remote access telnet protocol, leading one to believe that the developers wanted to use something already existing.

The FTP control connection is set up in the usual client–server fashion. That is, an FTP server process (such as ftpd) is listening for clients' connection requests. The number of simultaneous clients an FTP server can accept varies and is usually a configurable parameter, but limits well above 100 are not unusual.

The FTP server requires a log-in from the user, and in many cases servers will allow a special log-in for *anonymous FTP.* The user is supposed to use their email address as a password, a primitive auditing measure. Anonymous FTP implementations used to allow users to simply press Enter and leave the anonymous password field blank, but many FTP implementations now demand at least something at the password prompt. Some do not allow more creative substitutes for an email address, and many FTP servers check for things such as the presence of dots and the at sign (@) to try to enforce some semblance of honesty. In many cases, the FTP server will accept a similar term such as `guest` or `visitor`. The point behind anonymous FTP is that users are not required to have a valid user ID or password on the remote system in order to be able to access files in some directories.

Of course, there are file areas on the FTP server that should only be accessed by authenticated users of the remote system. Private IDs can be combined with anonymous FTP to protect certain areas of the file system while allowing public access to others. Of course, this does not stop people from *trying* to access files they had no business seeing, but if the file system permissions are set up correctly (or at all), FTP is highly secure. However, the best way to prevent access to sensitive files is *not* to put them on an FTP server with public access in the first place.

The well-known port of the control connection is TCP port 21. The client runs the FTP client program and uses an ephemeral port to begin the interaction with the server. This connection asks for the user ID and password, anonymous or not, and is nothing more than a normal remote log-in session using the Telnet application.

Once logged in, the user is placed in a default file system directory. Navigation outside this directory might be permitted, but usually there's a good reason to direct a user to this particular directory, and thus outside access should be unnecessary.

FTP Commands and Reply Codes

Users are sometimes surprised to see that FTP employs a very rich protocol all by itself. When run in interactive mode from the command line, FTP supplies its own prompt (like DNS) and supplies users with return codes for everything they type in.

The client and server have a conversation over the control connection, with the user at the client typing simple commands and sending them to the server process over the control connection. Some of the more common and helpful FTP commands are outlined in Table 20.1. These are the commands users type. But FTP sends four-character representations of these commands. For example, a `get` is a `RETR` (retrieve) and a `put` is `STOR` (store).

The server receives the command, takes the appropriate action (if allowed), and returns a numeric reply code. The reply codes are translated by the FTP client into text that can be understood easily and displayed at the prompt. The displayed text can vary from system to system because each FTP client implementation is free to interpret the reply codes, within reason, and display that text to the user. The meanings of the first and second digits of the reply codes are outlined in Table 20.2.

The third digit adds details. For example, the reply code `500` means that there is a syntax error and an unrecognized command has been sent to the server. The reply

Table 20.1 Common FTP Commands[a]

Command	Meaning
Open	Create an FTP connection between the two hosts.
Close	Close an FTP connection between two hosts.
Bye	End the FTP session.
Get	Retrieve a remote file from the remote host.
Put	Store a file on the remote host.
Mget	Get multiple files using wildcards (for example, mget a* fetches all files that being with the letter "a" in the current directory).
Mput	Put multiple files on the remote host using wildcards.
Glob	Enable wildcard interpretation. This is usually on by default.
Ascii	The file transferred is in ASCII representation (a common default).
Binary	The file is in image (binary) format (sometimes the default), and is useful for programs and formatted word processing files.
Cd	Change the directory on the remote host.
Dir	Get a directory listing from the remote host.
Ldir	Get a directory listing from the local host.
Hash	Display hash marks (dots) to show file transfer progress.

[a] *These commands are not part of the FTP protocol.*

code `501` means the syntax error is in the command arguments. If the reply code generates more than one line at the client (for example, if the valid arguments are listed), the reply code appears on the first line with a hyphen and is repeated at the end of the text.

The user then can type in another command. Common FTP replies, including the text that could be displayed with them, are:

- 125 Data connection open and transfer starting
- 200 Command okay
- 214 Help message (text follows)
- 331 User name okay, password required
- 425 Unable to open data connection
- 452 Error writing file
- 500 Command syntax error
- 501 Argument syntax error

Sessions end with the user typing `bye` or `quit` at the FTP prompt. The server should respond with a `221` reply, usually displayed as `221 Goodbye`. In some cases, the server

Table 20.2 FTP Protocol Reply Codes

Reply	Meaning
1xx	Positive response, but preliminary. Action begun, but wait for another reply before sending further commands.
2xx	Positive completion. New commands can be sent.
3xx	Positive response, but intermediate. Command accepted, but another command is required to complete the action.
4xx	Negative reply, but transient. Action did not take place, but the condition is temporary and the same command can be used again.
5xx	Negative reply, permanent. Action did not take place, and cannot be done. The command should not be sent again in that form.
x0x	Syntax error.
x1x	Information.
x2x	Reply refers to control or data connections.
x3x	Reply refers to authenticating and accounting commands, such as login.
x4x	Unspecified.
x5x	File system status information.

simply disappears, and one client we've used groused in the session log `You could at least say Goodbye`. But it is a sign of the robustness and stability of FTP that such breaches of protocol seldom mean that things do not work properly overall.

One advantage of running FTP from the command line instead of from a GUI is that the user can type in the entire array of FTP commands, which typically number 50 or more. GUI point-and-click clients can be prettier and easier, but do not always implement the full suite of FTP commands. (Some of the commands are seldom used or necessary today, such as `glob`, but might come in handy in certain situations.)

FTP Data Transfers

At some point in the FTP conversation between client and server port 21, the user will use a command that will trigger a file transfer. The transfer might not be the actual file itself, such as with `get` or `put`. Often, the user requests a file directory listing from the present working directory on the server with the `dir` command, usually to ensure that the desired file is there or to check the spelling after the first transfer attempt has failed. These actions require the server to set up an FTP *data connection.* (The control connection is just a Telnet remote access session and is inappropriate for bulk data transfer anyway.) The FTP model of control and data connections is shown in Figure 20.8.

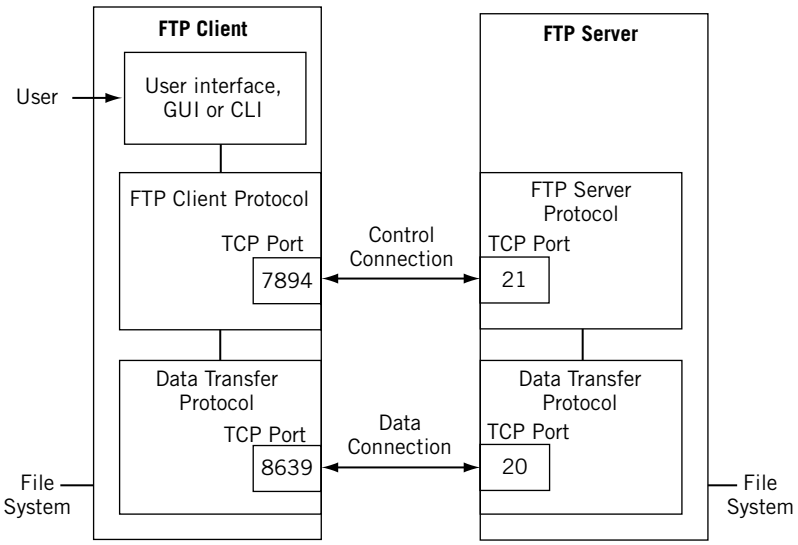


FIGURE 20.8

FTP control and data connections, showing how both are used in an FTP application.

Consider what happens when a user at an FTP client types in the dir command to receive a list of the contents of the remote host's directory. This requires the establishment of a data connection on the part of the server. The server normally uses well-known TCP port 20 as the server end of the data connection. But how does the client know which ephemeral port to listen on for the data?

The server sends an FTP `PORT` command over the control connection to the client with this information. This tells the client which port should be used at the client end for the data connection. So that there is no misunderstanding, the server includes the client's IP address as well. Thus, the command really supplies socket information. The PORT command is sent over the control connection and is formatted as if it were data to appear on a Telnet terminal, including control characters such as `\n` (new line).

The port number is expressed as two independent numbers. The first is multiplied by 256 and added to the second (which must be in the range 0–255) to give the client's port number. So, if the PORT command ends with the numbers `14, 234` (excluding the control characters) the port number the client should use for the data connection is 3818 ($14 \times 256 = 3584 + 234 = 3818$).

The client issues a passive open on port 3818, and the FTP server now sends a TCP SYN message to open the TCP session and send the `dir` listing as requested. The server usually closes the data connection as soon as the transfer is complete.

The control connection process of obtaining a simple `dir` listing from a remote FTP server is shown in Figure 20.9. Note that the client issues FTP commands and the server replies with codes.

The activity on the data connection is shown in Figure 20.10. Although in many cases the data connection uses well-known port 20 on the server, it does not have to.

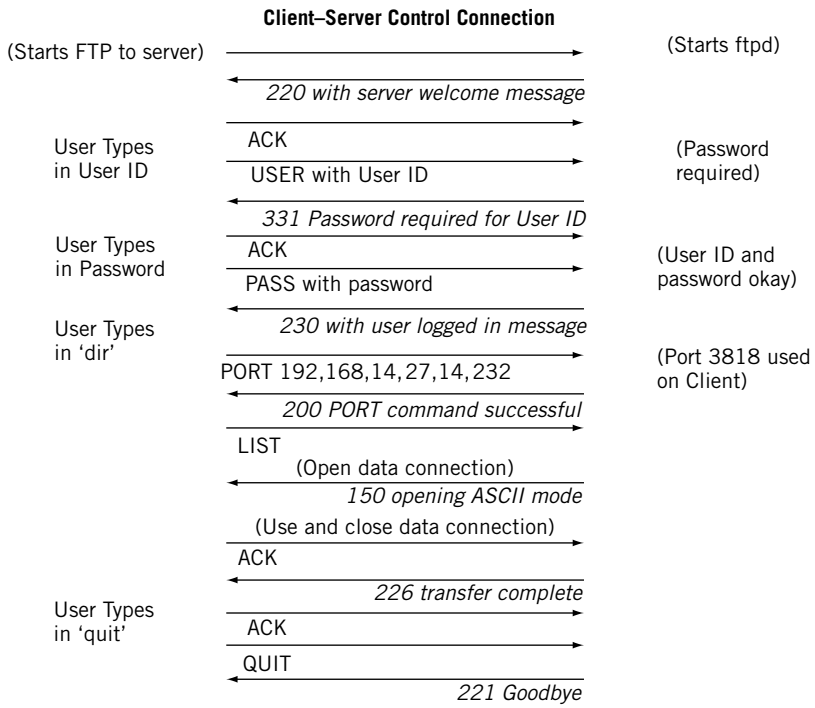


FIGURE 20.9

FTP control connection, showing how a directory listing proceeds.

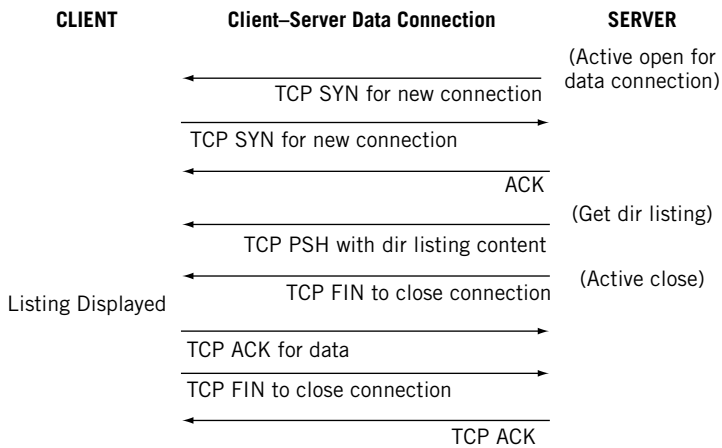


FIGURE 20.10

FTP data connection. The connection does not have to use port 20 on the server.

Passive and Port

Using the PORT command is not the only way the port used for the FTP data connection is determined. Today, the PORT command is considered in many cases to be an unacceptable security risk to an organization. This is because the PORT command requires an external FTP server to open a connection *to* an internal client. It is possible for a firewall to support incoming TCP connections for FTP, but with the common use of network address translation (see Chapter 27) it is simpler to use passive. (In larger installations using firewalls and NAT, collisions among the incoming port numbers are common anyway.)

FTP Passive

FTP supports two different methods of data connection establishment. In the normal *active* mode using PORT, the server (1) initiates the data connection, then (2) the client asks for a data transfer and (3) the client responds. In *passive* mode (PASV), the client tells the server that the client will initiate the data connection and the server responds. Passive mode allows the transfer to proceed when modern client devices are prohibited from accepting incoming data connections.

Consider the implication for a user sitting at a client host on a corporate LAN. We haven't talked about security in any detail, but in many cases the company will employ a *firewall* between internal LANs and the external world of the Internet. The firewall's job is to prevent malicious hackers or their code from attacking the hosts on the internal network.

One of the ways firewalls do this is to prevent any outside devices from establishing TCP connections to any internal client hosts on the LAN (publicly accessed servers are typically isolated, physically and logically, from purely internal hosts). Hosts accepting outside connections are seen, from the firewall's perspective, as vulnerable to any number of malicious worms or viruses. Many inexpensive firewalls also see an external FTP server's attempt to establish a TCP data connection to the client as a potential hostile attack. This attempt is blocked, and the transfer fails.

The PASV command reverses the procedure, and lets the *client* open the data connection to the server. Figure 20.11 shows the major difference between a client using the POST and PASV commands to initiate a data transfer. In both cases, the client uses port 4122 for the data connection. However, in active mode the *server* initiates the data connection and uses well-known port 20. In passive mode, the client initiates the data connection and listens on port 2020 instead of 20 for the connection.

However, all might still not be well. Many firewalls will not allow internal hosts to open connections to external ports that are not well known. After all, the malicious user could be on the local LAN and attacking someone else remotely. So, even when PASV is used the data connection set up might still fail.

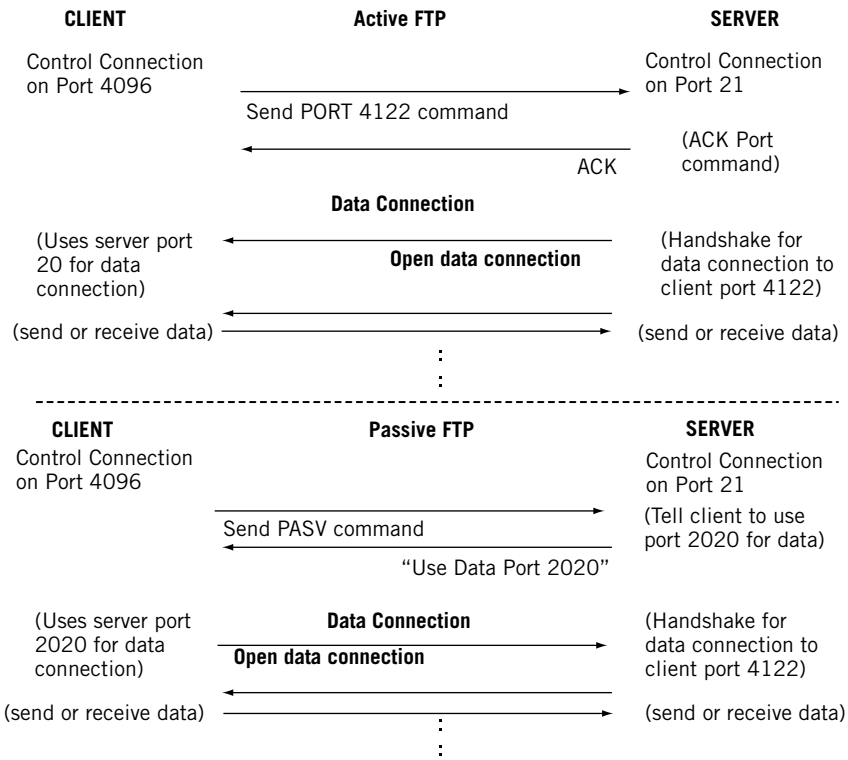


FIGURE 20.11

FTP active and passive. Note which side opens the data connection and which ports are used in each case.

More state-of-the-art firewalls will look at more than just TCP or UDP headers and can figure out that an FTP session is in progress. Many will only allow ports from a certain preconfigured pool to be used, but there is a lot of variation in implementation.

RFC 2428 defines the `EPRT` and `EPSV` commands to be used when IPv6 addresses and NAT is in use. Some FTP implementations use these forms of `PORT` and `PASS` by default. Network address translation can be particularly harsh on FTP because addresses can change. Some applications, such as FTP, send IP address and protocol ports inside messages as data. Unless NAT can change the addresses in the data stream to agree with its other changes, the application will fail. We'll talk more about NAT in a later chapter, but a full discussion of the interplay of NAT and FTP is beyond the scope of this book.

Sometimes the FTP application tries to get into the act and imposes certain conventions on the user. One FTP implementation insists on using `PASV` when it finds that private IPv4 addresses are being used, presumably because private addresses are only

used behind a firewall or when NAT is used. This particular form of FTP also insists that the user enter the public "WAN" address space used, which can be problematic when a purely private TCP/IP network such as the Illustrated Network is being used! (Needless to say, this application was not very useful on the Illustrated Network.)

File Transfer Types

What about the actual files that can be transferred from server to client or from client to server? The original FTP specification listed multiple options as to file type, embedded control characters, structure, and transmission mode. In those days, there were many types of computer architectures. Today, those choices usually boil down to exactly two: `ascii` and `binary`. Either one can be the implementation default, but as time goes on, pure text files using ASCII are becoming rarer and rarer, whereas files with executable code and embedded HTML formatting are becoming more and more common. FTP helpfully puts in line formatting control characters if they are missing when performing an `ascii` transfer. Naturally, this renders code files completely useless (although many newer FTP-based applications make this much less of a concern).

Unless there is a compelling reason to do otherwise, most FTP transfers are better off using `binary` (the file is transferred as a string of bits, and FTP makes no effort to figure out what they mean). This doesn't mean that the transferred file will be useful, but it has a better chance than a file of program code transferred as a text note with `ascii`.

When Things Go Wrong

There is a huge benefit to keeping FTP data transfers off the control connection. The use of two connections allows users to abort a file transfer that is unintended or out of control (a misformed `mget` is usually the culprit). When the client is storing a file on a server, the use of the control connection is straightforward: The client stops sending data and sends an `ABOR` command to the sender on the control connection. The interrupt key is usually `cntl-C`, but others are possible depending on operating system. The `ABOR` command is sent as urgent TCP data to make sure it is handled promptly by the server.

When the server receives the `ABOR` command on the control connection, it should respond with 426 (transfer aborted) and 226 (abort successful) messages. The data transfer might continue sending data, and typically does, but the client will not acknowledge it and ignores everything received after the user abort.

There are only a few other things that can go wrong with FTP. A common mistake is to transfer binary files as text, and some FTP servers will warn the user if the file extension seems to indicate this might be going to happen. Other servers assume that users know what they are doing and simply perform the transfer.

There are two other parameters dealing with file transfer in FTP that can be changed and might cause problems when multiple files are transferred without restoring the settings. One is the *file-structure*. A transfer can use *file-structure* (the name is unfortunate)

or *record-structure*. File-structure, the usual default, makes no assumptions about the file at all and simply views the content as a string of bytes. Record-structure, rarely used today, means that there is a record format to the file and is set by sending the `STRU R` command to the other host.

Even when the record-structure is set for the transfer, the actual formatting of the data depends on another setting—this one is called the *transmission mode*. Modes can be *stream* (the typical default), *block*, and *compressed*. The three modes combine with the file-structure to give four types of file transfer formatting.

Stream mode with file-structure—The file is set as a stream of bytes, and TCP provides data integrity. No headers or delimiters are inserted into the data stream, and the end of the transferred file is only indicated by closing the data connection normally. This is the most common way in which FTP works on the Internet today.

Stream mode with record-structure—The file is sent as a string of records, each one delimited by a 2-byte End of Record (EOR) control code (`0xFF01`). An End of File (EOF) code, `0xFF02` (or sometimes `0xFF03`), is used to indicate the end of the file to the receiver.

Block mode—The file is sent as a series of data blocks. Each block begins with a 3-byte header containing some descriptor flags and a 2-byte length field giving the block byte count. Flags are used to indicate EOR, EOF, and restart.

Compressed mode—Rarely supported today because modern compression methods have superseded this primitive function. The file is sent after removing repeated string of bytes. Today, files are compressed outside FTP and sent as `binary` data.

Finally, many FTP server implementations routinely check the domain name of the client to make sure it is valid before allowing the connection. Reverse DNS, as this is called, is not a robust security feature, and at times has caused problems as well on the network. Hackers can easily use phony IP addresses, the theory goes, but it's more difficult (and foolish) to map it to a public domain name and distribute the information by registering on the public DNS. This was a problem with some early Illustrated Network file transfers because no DNS was running on the network at all, and even when it was no Illustrated Network domain names were registered on the Internet. But "dumber" FTP versions worked just fine with only IP addresses.

FTP COMMANDS

One of the things that surprises people when they examine traces of FTP activity is that the FTP commands sent and received by client and server are not the same as the ones entered by the user at the client. We've already looked at some examples (cntl-C sends an `ABORT`), but maybe it's a good idea to look at them in more detail.

Table 20.3 FTP Commands for File Server Access with Meaning and Parameters

Command	Meaning	Parameter(s)
USER	User ID	User ID
PASS	User password	Password itself
ACCT	Provide an account for charging purposes	Account ID
REIN	Reinitialize to the start state	None
QUIT	End and log out	None
ABORT	Abort previous command and any file transfer	None

Table 20.4 FTP Commands for Remote Server File Management with Meaning and Parameters

Command	Meaning	Parameter(s)
CWD	Change to another directory	Directory path
CDUP	Change to the parent directory	None
DELE	Delete a file	File name
LIST	List file information	None, or directory name, or list of files
MKD	Make a directory	Directory name
NLST	List the files in a directory	None for current directory, or name
PWD	Show the name of the current working directory	None
RMD	Remove a directory	Directory name
RNFR	Rename a file (references current name)	Current file name
RNTO	Rename a file (references new name)	New file name
SMNT	Mount a different file system	File system identifier
ABORT	Abort previous command and any file transfer	None

Clients and servers do not have to implement all of the FTP commands, which are often added to. What happens if a server requires the user at the client to use an FTP command the client implementation does not support? A thorough client will implement the `quote` user command, which lets the user enter the exact formal command (and any parameters) necessary to continue. The input is then sent over the control connection exactly as entered.

The six FTP commands that control a user's access to a remote file server are outlined in Table 20.3. The 11 FTP commands that control a user's file access and management functions on the remote file server are outlined in Table 20.4. The working directory is the current directory.

Table 20.5 FTP Commands for Transfer Parameters, with Meaning and Parameters

Command	Meaning	Parameter(s)
TYPE	Identify the file type for transfer	A (ASCII), E (EBCDIC), I (binary image), N (nonprint), T (telnet), C (ASA)
STRU	File structure	F (file) or R (record)
MODE	Format used for transmission	S (stream), B (block), C (compressed)

Table 20.6 FTP Commands for File Transfer, with Meaning and Parameters

Command	Meaning	Parameter(s)
ALLO	Allocate enough space for the data to come	Integer number of bytes
APPE	Append a local file to the remote file	File names
EPSV	The extended version (RFC 2428) of the PASV command, used for IPv6 and NAT	IP address and port
EPRT	The extended version (RFC 2428) of the PORT command, used for IPv6 and NAT	IP address and port
PASV	Supply the network address and port number that will be used for the data connection initiated by the client	IP address and port
PORT	Supply the network address and port number that will be used for the data connection initiated by the server	IP address and port
REST	Identify a restart marker (followed by the transfer command to be restarted)	Marker value
RETR	Get (retrieve) a file	File name(s)
STOR	Put (store) a file	File name(s)
STOU	Create a version of the file with a unique name (store unique)	File name

The three FTP commands that set the type, structure, and mode of the file transfer are outlined in Table 20.5. The 10 FTP commands that actually control the file transfer are outlined in Table 20.6. Finally, the five FTP commands outlined in Table 20.7 supply useful information to the user.

Variations on a Theme

Few people use the command line interface for FTP unless they have to. However, it is common to use the CLI for instructional purposes (as done here). But today almost all FTP client software, and many servers, use GUI interfaces to let users simply point and

Table 20.7 FTP Commands for User Information, with Meaning and Parameters

Command	Meaning	Parameter(s)
HELP	Gives information about server implementation	None
NOOP	Request "OK" reply from server	None
SITE	Used in the popular WU-FTP implementation from Washington University (used in many Linux versions) to engage server-specific commands not in the FTP standard	None
SYST	Requests that the server identify its OS version	None
STAT	Request connection status and parameter information from server	None

click at directories and files and effect a transfer. Almost all still allow users to watch the interplay between mouse strokes and FTP commands and response codes, but few pay attention to them unless things go wrong.

GUI implementations of FTP tend to be much more sophisticated than their CLI cousins, especially when it comes to security variations. The heavy use of security on modern networks has spawned many variations of the simple FTP control and data connection process. Most of these variations have to do with how the user ID and password are packaged and sent from client to server, but some are more far-reaching than that. Many commercial FTP server implementations can be set up to function in any of the following environments:

- Simple FTP
- FTP over Secure Sockets Layer and Transport Layer Security (SSL/TLS), using implicit encryption
- FTP over SSL/TLS using explicit encryption
- FTP over TLS directly, using explicit encryption
- FTP bypassing the firewall

We'll have much more to say about these security variations later in this book. There is also Secure FTP (SFTP), a feature of Secure Shell 2 (SSH2). But this is a completely different protocol than FTP, as we'll see in Chapter 25 (on SSH).

A Note on NFS

If TCP/IP is indeed for everything, an employee at a branch bank should be able to use common TCP/IP applications to change a customer's information in the central bank's database. However, it makes no sense at all to access the master account file, transfer a copy of it to the branch host, update it, and then load it back up to the central location. Not only does this method transfer masses of information *not* needed, but it prevents (hopefully) anyone else from updating any other customer record at the same time.

Many applications don't want or need remote file *transfer*. They just need remote file *access*, usually to a particular record or even field. This is the idea behind the Network File System (NFS), pioneered by Sun Microsystems. NFS allows local file systems to be accessed by remote users as if they were local users and is a nice illustration of the power and utility of the socket interface.

NFS is actually part of an overall system that includes an extension of the socket concept known as remote procedure calls (RPCs). RPCs are a more sophisticated way of handling basic programming subroutine (or function) calls by allowing the subprogram (the procedure) to be called on a remote system across a network (hence the term *remote procedure call*).

RPCs do not use well-known ports. RPC server processes handle RPC client requests for server connections by *dynamically mapping* the server ports. In dynamic mapping, all connection requests handled by TCP go to *one* server process running at the application layer instead of several. This server process is capable of dynamically starting up the correct port server application process and allowing the TCP protocol to grant the connection. The single server application process running under dynamic mapping is known as the *port mapper*. These port mappers (usually run as the `rpcbind` process) are very common on most Unix implementations of TCP/IP.

Another part of the NFS is the External Data Representation (XDR) standard, a way of defining data types in terms of standard formats. The point is to allow remote file access between different platforms, from Unix to Windows to MACs and even more. NFS has been a part of the overall TCP/IP standardization process since 1998.

QUESTIONS FOR READERS

Figure 20.12 shows some of the concepts discussed in this chapter and can be used to answer the following questions.

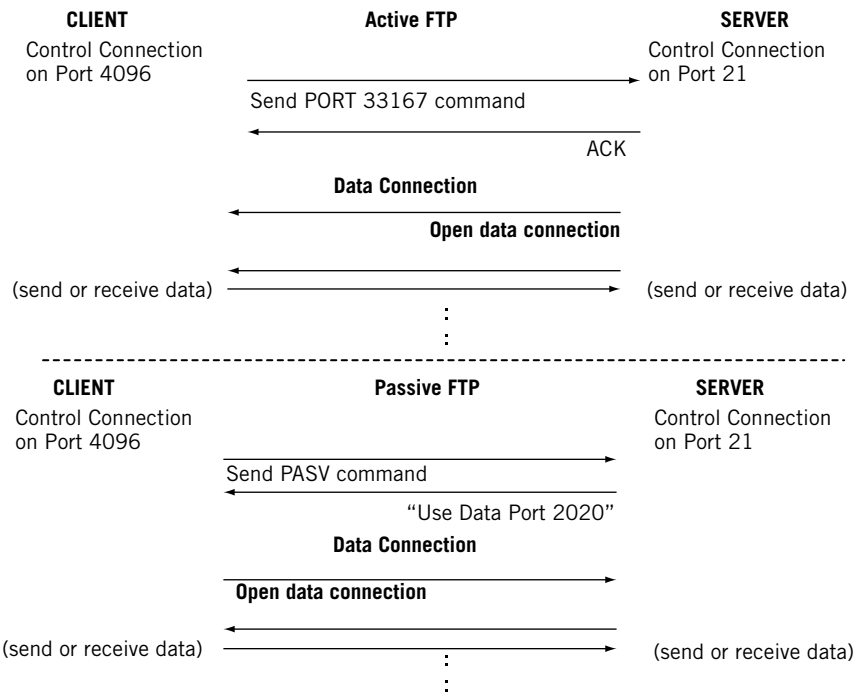


FIGURE 20.12

Simplified view of active and passive data transfer modes.

1. Who initiates the data connection in active and passive mode, respectively?
2. In the figure, for active mode what port will the client use on the server for data transfer?
3. In the figure, for passive mode what port will the client use on the server for data transfer?
4. In the figure, what port will the client use for the data connection in active mode?
5. In the figure, what port will the client use for the data connection in *passive* mode? How does the server know what it is?