CHAPTER

Multiplexing and Sockets

12

What You Will Learn

In this chapter, you will learn about how multiplexing (and demultiplexing) and sockets are used in TCP/IP. We'll see how multiplexing allows many applications can share a single TCP/IP stock process.

You will learn how layer and applications interact to make multiplexing and the socket concept very helpful in networking. We'll use a small utility program to investigate sockets and illustrate the concepts in this chapter.

Now that we've looked at UDP and TCP in detail, this chapter explores two key concepts that make understanding how UDP and TCP work much easier: *multiplexing* and *sockets*. Technically, the term should be "multiplexing and demultiplexing," but because mixing things together makes little sense unless you can get them back again, most people just say "multiplexing" and let it go at that.

Why is multiplexing necessary? Most TCP/IP hosts have only one TCP/IP stack process running, meaning that every packet passing into or out of the host uses the same software process. This is due to the fact that the hosts usually have only one network connection, although there are exceptions. However, a host system typically runs many (technically, if other systems can access them, the host system is a server). All these applications share the single network interface through multiplexing.

LAYERS AND APPLICATIONS

Both the source and destination port numbers, each 16 bits long, are included as the first fields of the TCP or UDP segment header. Well-known ports use numbers between 0 and 1023, which are reserved expressly for this purpose. In many TCP/IP implementations, there is a process (usually `inetd` or `xinetd`, the "Internet daemon") that listens for *all* TCP/IP activity on an interface. This process then launches to FTP or other application processes on request, using the well-known ports as appropriate.

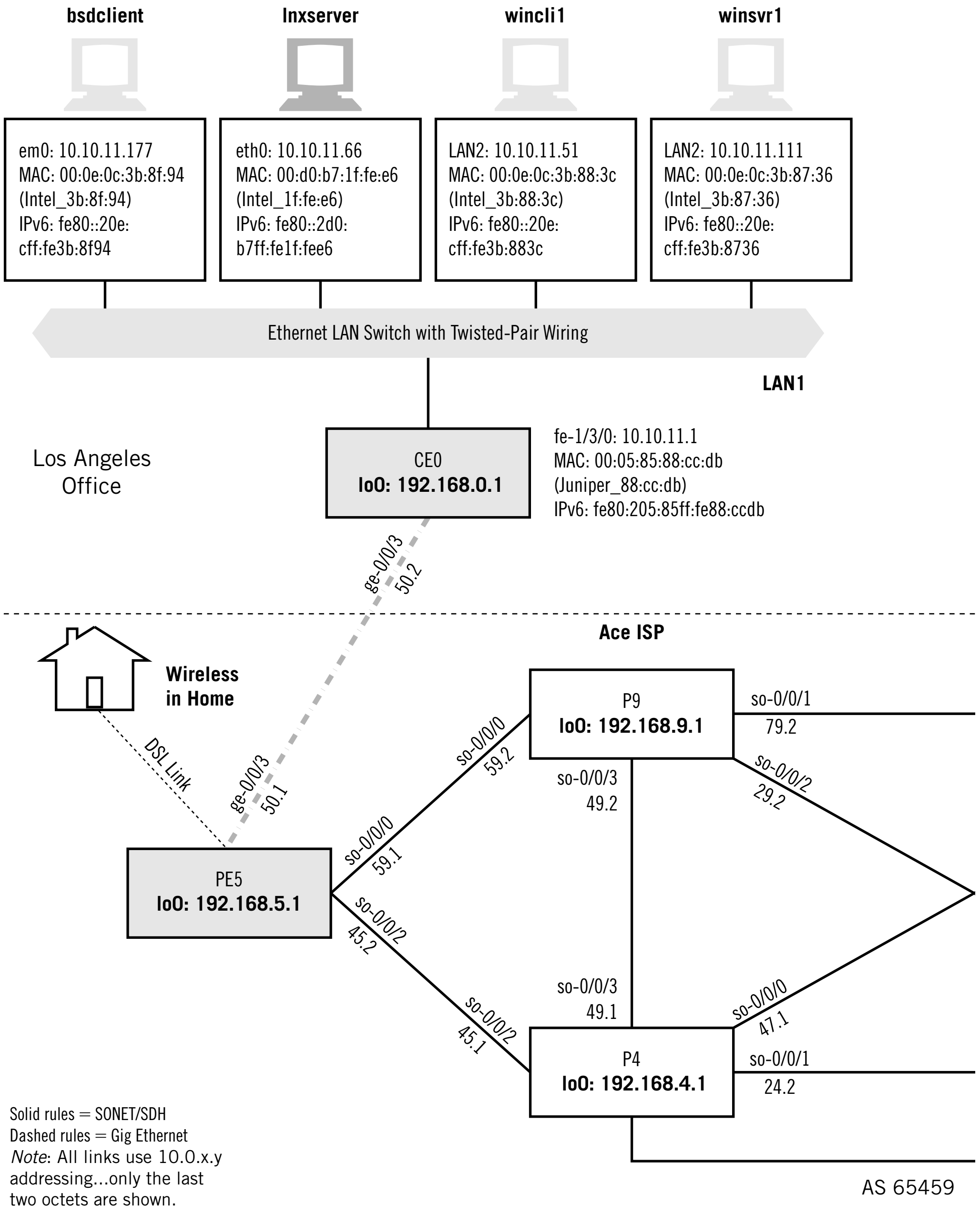
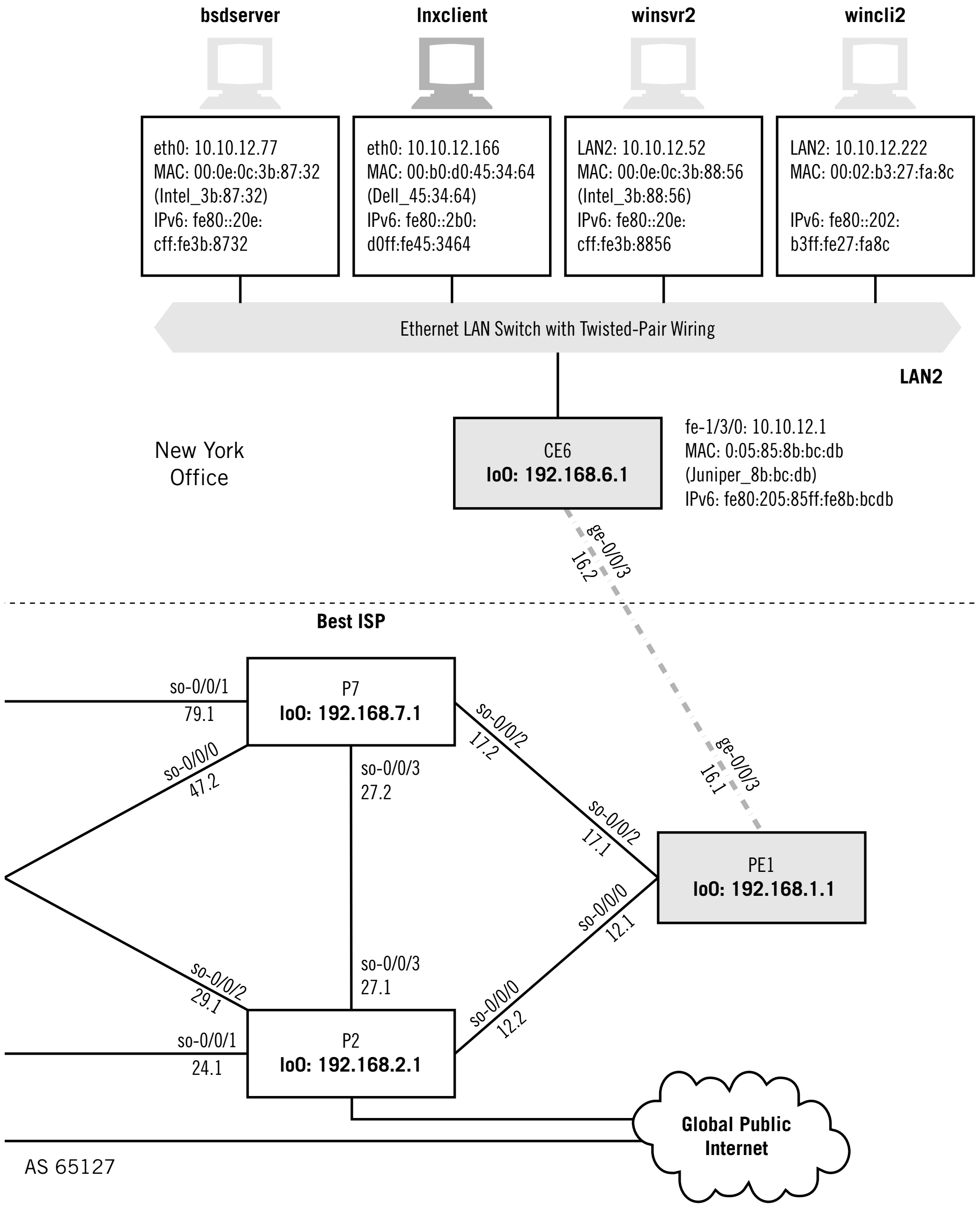


FIGURE 12.1

Sockets between Linux client and server, showing the devices used in this chapter to illustrate socket operation.


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
MAC: 00:b0:d0:45:34:64
(Dell_45:34:64)
IPv6: fe80::2b0:
d0ff:fe45:3464
LAN2: 10.10.12.52
MAC: 00:0e:0c:3b:88:56
(Intel_3b:88:56)
IPv6: fe80::20e:
cff:fe3b:8856
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
ge-0/0/3
16.1
so-0/0/0
47.2
so-0/0/3
27.2
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

However, the well-known server port numbers can be *statically mapped* to their respective application on the TCP/IP server, and that's how we will explore them in this introduction to sockets. With static mapping, the DNS (port number 53) or FTP (port number 21) server processes (for example) must be running on the server at all times in order for the server TCP protocol to accept connections to these application form clients. Things are more complex when both IPv4 and IPv6 are running, but this chapter considers the situation for IPv4 for simplicity.

This chapter will be a little different than the others. Instead of jumping right in and capturing packets and then analyzing them, the socket packet capture is actually the whole point of the chapter. So we'll save that until last. In the meantime, we'll develop a socket-based application to work between the `lnxclient` (`10.10.12.166` on LAN2) and `lnxserver` (`10.10.11.66` on LAN1), as shown in Figure 12.1.

THE SOCKET INTERFACE

Saying that applications share a single network connection through multiplexing is not much of an explanation. *How* does the TCP/IP process determine the source and destination application for the contents of an arriving segment? The answer is through sockets. Sockets are the combination of IP address and TCP/UDP port number. Hosts use sockets to identify TCP connections and sort out UDP request–response pairs, and to make the coding of TCP/IP applications easier.

The server TCP/IP application processes that "listen" through passive opens for connection requests use well-known port numbers, as already mentioned. The client TCP/IP application processes that "talk" through active opens and make connection requests must choose port numbers that are not reserved for these well-known numbers. Servers listen on a socket for clients talking to that socket. There is nothing new here, but sockets are more than just a useful concept. The *socket interface* is the most common way that application programs interact with the network.

There are several reasons for the socket interface concept and construct. One reason has already been discussed. Suppose there are two FTP sessions in progress to the same server, and both client processes are running over the same network connection on a host with IP address `192.168.10.70`. It is up to the client to make sure that the two processes use different client port numbers to control the sessions to the server. This is easy enough to do. If the clients have chosen client port numbers `14972` and `14973`, respectively, the FTP server process replies to the two client sockets as `192.168.10.79:14972` and `192.168.10.70:14973`. So the sockets allow simultaneous file transfer sessions to the same client from the same FTP server. If the client sessions were distinguished only by IP address or port number, the server would have no way of uniquely identifying the client FTP process. And the FTP server's socket address is accessed by all of the FTP clients at the same time without confusion.

Now consider the server shown in Figure 12.2. Here there is a server that has more than one TCP/IP interface for network access, and thus more than one IP address. Yet these servers may still have only one FTP (or any other TCP/IP application) server process

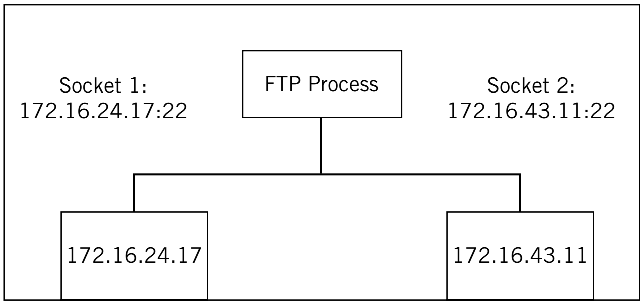


FIGURE 12.2

The concept of sockets applied to FTP. Note how sockets allow a server with two different IP addresses to access the FTP server process using the same port.

running. With the socket concept, the FTP server process has no problem separating client FTP sessions from different network interfaces because their socket identifiers will differ on the server end. Since a TCP connection is always identified by both the client and server IP address and the client and server port numbers, there is no confusion.

This illustrates the sockets concept in more depth, but not the use of the socket interface in a TCP/IP network. The socket interface forms the boundary between the application program written by the programmer and the network processes that are usually bundled with the operating system and quite uniform compared to the myriad of applications that have been implemented with programs.

Socket Libraries

Developers of applications for TCP/IP networks will frequently make use of a *sockets library* to implement applications. These applications are not the standard "bundled" TCP/IP applications like FTP, but other applications for remote database queries and the like that must run over a TCP/IP network. The sockets library is a set of programming tools used to simplify the writing of application programs for TCP or UDP. Since these "custom" applications are not included in the regular application services layer of TCP/IP, these applications must interface directly with the TCP/IP stack. Of course, these applications must also exist in the same client–server, active–passive open environment as all other TCP/IP applications.

The socket is the programmer's identifier for this TCP/IP layer interface. In Unix environments, the socket is treated just like a file. That is, the socket is created, opened, read from, written to, closed, and deleted, which are exactly the same operations that a programmer would use to manipulate a file on a local disk drive. Through the use of the socket interface, a developer can write TCP/IP networked client–server applications without thinking about managing TCP/IP connections on the network.

The programmer can use sockets to refer to any remote TCP/IP application layer entity. Many developers use socket interfaces to provide "front-end" graphical interfaces

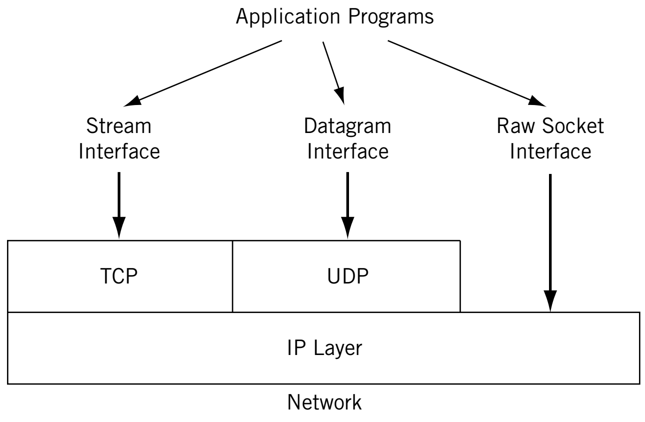


FIGURE 12.3

The three socket types. Note that the raw socket interface bypasses TCP and UDP. (The socket program often builds its own TCP or UDP header.)

to common remote TCP/IP server processes such as FTP. Of course, the developers may choose to write applications that implement both sides of the client–server model.

The socket can interface with either TCP (called a "stream" socket), UDP (called a "datagram" socket), or even IP directly (called the "raw" socket). Figure 12.3 shows the three major types of socket programming interfaces. There are even socket libraries that allow interfaces with the frames of the network access layer below IP itself. More details must come from the writers of the sockets libraries themselves, since socket libraries vary widely in operational specifics.

TCP Stream Service Calls

When used in the stream mode, the socket interface supplies the TCP protocol with the proper service *calls* from the application. These service calls are few in number, but enough to completely activate, maintain, and terminate TCP connections on the TCP/IP network. Their functions are summarized in the following:

OPEN—Either a passive or active open is defined to establish TCP connections.

SEND—Allows a client or server application process to pass a buffer of information to the TCP layer for transmission as a segment.

RECEIVE—Prepares a receive buffer for the use of the client or server application to receive a segment from the TCP layer.

STATUS—Allows the application to locate information about the status of a TCP connection.

CLOSE—Requests that the TCP connection be closed.

ABORT—Asks that the TCP connection discard all data in buffers and terminate the TCP connection immediately.

These commands are invoked on the application's behalf by the socket interface, and therefore are not seen by the application programmer. But it is always good to keep in mind that no matter how complicated a sockets library of routines might seem to a programmer, at heart the socket interface is a relatively simple procedure.

THE SOCKET INTERFACE: GOOD OR BAD?

However, the very simplicity of socket interfaces can be deceptive. The price of this simplicity is isolating the network program developers from any of the details of how the TCP/IP network actually operates. In many cases, the application programmers interpret this "transparency" of the TCP/IP network ("treat it just like a file") to mean that the TCP/IP network really does not matter to the application program at all.

As many TCP/IP network administrators have learned the hard way, nothing could be further from the truth. Every segment, datagram, frame, and byte that an application puts on a TCP/IP network affects the performance of the network for everyone. Programmers and developers that treat sockets "just like a file" soon find out that the TCP/IP network is not as fast as the hard drive on their local systems. And many applications have to be rewritten to be more efficient just because of the seductive transparency of the TCP/IP network using the socket interface.

For those who have been in the computer and network business almost from the start, the socket interface controversy in this regard closely mirrors the controversy that erupted when COBOL, the first "high-level" programming language, made it possible for people who knew absolutely nothing about the inner workings of computers to be trained to write application programs. Before COBOL, programmers wrote in a low-level assembly language that was translated (assembled) into machine instructions. (Some geniuses wrote directly in machine code without assemblers, a process known as "bare metal programming.")

Proponents then, as with sockets, pointed out the efficiencies to be enjoyed by freeing programmers from reinventing the wheel with each program and writing the same low-level routines over and over. There were gains in production as well—programmers who wrote a single instruction in COBOL could rely on the compiler to generate about 10 lines of underlying assembly language and machine code. Since programmers all wrote about the same number of lines of code a day, a 10-fold gain in productivity could be claimed.

The same claims regarding isolation are often made for the socket interface. Freed from concerns about packet headers and segments, network programmers can concentrate instead on the real task of the program and benefit from similar productivity gains. Today, no one seriously considers the socket interface to be an isolation liability, although similar claims of "isolation" are still heard when programmers today can generate code by pointing and clicking at a graphical display in Visual Basic or another even higher level "language."

The “Threat” of Raw Sockets

A more serious criticism of the socket interface is that it forms an almost perfect tool for hackers, especially the raw socket interface. Many network security experts do not look kindly on the kind of abuses that raw sockets made possible in the hands of hackers.

Why all the uproar over raw sockets? With the stream (TCP) and datagram (UDP) socket interfaces, the programmer is limited with regard to what fields in the TCP/UDP or IP header that they can manipulate. After all, the whole goal is to relieve the programmer of addressing and header field concerns. Raw sockets were originally intended as a protocol research tool only, but they proved so popular among the same circle of trusted Internet programmers at the time that use became common.

But raw sockets let the programmer pretty much control the entire content of the packet, from header to flags to options. Want to generate a SYN attack to send a couple of million TCP segments with the SYN bit sent one after the other to the same Web site, and from a phony IP address? This is difficult to do through the stream socket, but much easier with a raw socket. Consequently, this is one reason why you can find and download over the Internet hundreds of examples using TCP and UDP sockets, but raw socket examples are few and far between. Not only could users generate TCP and UPD packets, but even “fake” ICMP and traceroute packets were now within reach.

Microsoft unleashed a storm of controversy in 2001 when it announced support for the “full Unix-style” raw socket interface in Windows XP. Limited support for raw sockets in Windows had been available for years, and third-party device drivers could always be added to Windows to support the full raw socket interface, but malicious users seldom bestirred themselves to modify systems that were already in use. However, if a “tool” was available to these users, it would be exploited sooner or later.

Many saw the previous limited support for raw sockets in Windows as a blessing in disguise. The TCP/UDP layers formed a kind of “insulation” to protect the Internet from malicious application programs, a protective layer that was stripped away with full raw socket support. They also pointed out that the success of Windows NT servers, and then Windows 95/98/Me, all of which lacked full raw socket support, meant that no one *needed* full raw sockets to do what needed doing on the Internet. But once full raw sockets came to almost everyone’s desktop, these critics claimed, hackers would have a high-volume, but poorly secured, operating system in the hands of consumers.

Without full raw sockets, Windows PCs could not spoof IP addresses, generate TCP segment SYN attacks, or create fraudulent TCP connections. When taken over by email-delivered scripts in innocent-looking attachments, these machines could become “zombies” and be used by malicious hackers to launch attacks all over the Internet.

Microsoft pointed out that full raw sockets support was *possible* in previous editions of Windows, and that “everybody else had them.” Eventually, with the release of Service Pack 2 for Windows XP, Microsoft restricted the traffic that could be *sent* over the raw socket interface (receiving was unaffected) in two major ways: TCP data could not be sent and the IP source address for UDP data must be a valid IP address. These changes should do a lot to reduce the vulnerability on Windows XP in this regard.

Also, in traditional Unix-based operating systems, access to raw sockets is a privileged activity. So, in a sense the issue is not to hamper raw sockets, but to prevent unauthorized access to privileged modes of operation. According to this position, all raw socket restrictions do is hamper legitimate applications and form an impediment to effectiveness and portability. Restrictions have never prevented a subverted machine from spoofing traffic before Windows XP or since.

Socket Libraries

Although there is no standard socket programming interface, there are some socket interfaces that have become very popular for a number of system types. The original socket interface was developed for the 1982 version of the Berkeley Systems Distribution of Unix (BSD 4.1c). It was designed at the time to be used with a number of network protocol architectures, not just TCP/IP alone. But since TCP/IP was bundled with BSD Unix versions, sockets and TCP/IP have been closely related. A number of improvements have been made to the original BSD socket interface since 1982. Some people still call the socket interfaces "Berkeley sockets" to honor the source of the concept.

In 1986, AT&T, the original developers of Unix, introduced the Transport Layer Interface (TLI). The TLI interface was bundled with AT&T UNIX System V and also supported other network architectures besides TCP/IP. However, TLI is also almost always used with TCP/IP network interface. Today, TLI remains somewhat of a curiosity.

WinSock, as the socket programming interface for Windows is called, is a special case and deserves a section of its own.

THE WINDOWS SOCKET INTERFACE

One of the most important socket interface implementations today, which is not for the Unix environment at all, is the Windows Socket interface programming library, or WinSock. WinSock is a dynamic link library (DLL) function that is linked to a Windows TCP/IP application program when run. WinSock began with a 16-bit version for Windows 3.1, and then a 32-bit version was introduced for Windows NT and Windows 95. All Microsoft DLLs have well-defined application program interface (API) calls, and in WinSock these correspond to the sockets library functions in a Unix environment.

It is somewhat surprising, given the popularity of the TCP/IP protocol architecture for networks and the popularity of the Microsoft Windows operating system for PCs, that it took so long for TCP/IP and Windows to be used together. For a while, Microsoft (and the hardcover version of Bill Gates's book) championed the virtues of multimedia CD-ROMs over the joys of surfing the Internet, but that quickly changed when the softcover edition of the book appeared and Microsoft got on the Internet bandwagon (much to the chagrin of Internet companies like Netscape). In fairness to Microsoft, there were lots of established companies, such as Novell, that failed to foresee the rise of the Internet and TCP/IP and their importance in networking. There were several reasons for the late merging of Windows and TCP/IP.

TCP/IP and Windows

First, TCP/IP was always closely associated with the Unix world of academics and research institutions. As such, Unix (and the TCP/IP that came with it) was valued as an open standard that was easily and readily available, and in some cases even free. Windows, on the other hand, was a commercial product by Microsoft intended for corporate or private use of PCs. Windows came to be accepted as a proprietary, de facto standard, easily and readily available, but never for free. Microsoft encouraged developers to write applications for Windows, but until the release of Windows for Workgroups (WFW) these applications were almost exclusively "stand-alone" products intended to run complete on a Windows PC. Even with the release of Windows for Workgroups, the network interface bundled with WFW was not TCP/IP, but NetBIOS, a network interface for LANs jointly owned by IBM and Microsoft.

Second, in spite of Windows multitasking capabilities (the ability to run more than one process at a time), Windows used a method of multitasking known as "non-preemptive multitasking." In non-preemptive multitasking, a running process had to "pause" during execution on its own, rather than the operating system taking control and forcing the application to pause and give other processes a chance to execute. Unix, in contrast, was a preemptive multitasking environment. With preemptive multitasking, the Unix operating system keeps track of all running processes, allocating computer and memory resources so that they all run in an efficient manner. This system is characterized by more work for the operating system, but it is better for all the applications in the long run. Windows was basically a multitasking GUI built on top of a single-user operating system (DOS).

Sockets for Windows

The pressure that led to the development of the WinSock interface is simple to relate. Users wanted to hook their Windows-based PCs into the Internet. The Internet only understands one network protocol, TCP/IP. So WinSock was developed to satisfy this user need. At first the WinSock interface was used almost exclusively to Internet-enable Windows PCs. That is, the applications developed in those pre-Web days to use the WinSock interface were simple client process interfaces to enable Windows users to Telnet to Internet sites, run FTP client process programs to attach to Internet FTP servers, and so on. This might sound limited, but before WinSock, Windows users were limited to dialing into ports that offered asynchronous terminal text interfaces and performed TCP/IP conversion for Windows users.

There were performance concerns with those early Windows TCP/IP implementations. The basic problem was the performance of multitasked processes in the Microsoft Windows non-preemptive environment. Most TCP/IP processes, client or server, do not worry about when to run or when to pause, as the Unix operating system handles that. With Windows applications written for the WinSock DLL, all of the TCP/IP processes worried about the decision of whether to run or pause, since the Windows operating system could not "suspend" or pause them on its own. This voluntary giving up of execution time was a characteristic of Windows, but not of most TCP/IP implementations.

Also, Unix workstations had more horsepower than PC architectures in those early days, and the Unix operating system has had multitasking capabilities from the start. Originally, Unix required a whole minicomputer's resources to run effectively. When PCs came along in the early 1980s, they were just not capable of having enough memory or being powerful enough to run Unix effectively (a real embarrassment for the makers of AT&T PCs for a while). By the early 1990s, when the Web came along, early Web sites often relied on RISC processors and more memory than Windows PCs could even address in those days.

It is worth pointing out that most of these limitations were first addressed with Windows 95, the process continued with Windows NT, and finally Windows XP and Vista. Today, no one would hesitate to run an Internet server on a Windows platform, and many do.

SOCKETS ON LINUX

Any network, large or small, can use sockets. In this section, let's look at some socket basics on Linux systems.

We could write socket client and server applications from scratch, but the truth is that programmers hate to write anything from scratch. Usually, they hunt around for code that does something pretty close to what they want and modify it for the occasion (at least for noncommercial purposes). There are plenty of socket examples available on the Internet, so we downloaded some code written by Michael J. Donahoo and Kenneth L. Calvert. The code, which comes with no copyright and a "use-at-your-own-risk" warning, is taken from their excellent book, *TCP/IP Sockets in C* (Morgan Kaufmann, 2001).

We'll use TCP because there should be more efficiency derived from a connection-oriented, three-way handshake protocol like TCP than in a simple request–response protocol like UDP. This application sends a string to the server, where the server socket program bounces it back. (If no port is provided by the user, the client looks for well-known port 7, the TCP Echo function port.) First, we'll list out and compile my version of the client socket code (`TCPsocketClient` and `DieWithError.c`) on `lnxclient`. (Ordinarily, we would put all this is its own directory.)

```
[root@lnxclient admin]# cat TCPsocketClient.c
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), connect(), send(), and recv() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_addr() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define RCVBUFSIZE 32   /* Size of receive buffer */
```

```
void ErrorFunc(char *errorMessage);  /* Error handling function */

int main(int argc, char *argv[])
{
    int sock;                        /* Socket descriptor */
    struct sockaddr_in echoServAddr; /* Echo server address */
    unsigned short echoServPort;     /* Echo server port */
    char *servIP;                    /* Server IP address (dotted quad) */
    char *echoString;                /* String to send to echo server */
    char echoBuffer[RCVBUFSIZE];     /* Buffer for echo string */
    unsigned int echoStringLen;      /* Length of string to echo */
    int bytesRcvd, totalBytesRcvd;   /* Bytes read in single recv()
                                        and total bytes read */

    if ((argc < 3) || (argc > 4))    /* Test for correct number of
                                        arguments  */
    {
       fprintf(stderr, "Usage: %s <Server IP> <Echo Word> [<Echo Port>]\n",
               argv[0]);
       exit(1);
    }

    servIP = argv[1];       /* First arg: server IP address (dotted quad) */
    echoString = argv[2];   /* Second arg: string to echo */

    if (argc == 4)
        echoServPort = atoi(argv[3]); /* Use given port, if any */
    else
        echoServPort = 7;  /* 7 is the well-known port for the echo
                              service */

    /* Create a reliable, stream socket using TCP */
    if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");

    /* Construct the server address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));      /* Zero out
                                                            structure */
    echoServAddr.sin_family    = AF_INET;                /* Internet address
                                                            family */
    echoServAddr.sin_addr.s_addr = inet_addr(servIP);  /* Server
                                                            IP address */
    echoServAddr.sin_port   = htons(echoServPort);     /* Server port */

    /* Establish the connection to the echo server */
    if (connect(sock, (struct sockaddr *) &echoServAddr,
      sizeof(echoServAddr)) < 0)
       DieWithError("connect() failed");

    echoStringLen = strlen(echoString);        /* Determine input length */
    /* Send the string to the server */
    if (send(sock, echoString, echoStringLen, 0) != echoStringLen)
        DieWithError("send() sent a different number of bytes than expected");

    /* Receive the same string back from the server */
    totalBytesRcvd = 0;
    printf("Received: ");              /* Setup to print the echoed string */
    while (totalBytesRcvd < echoStringLen)
    {
        /* Receive up to the buffer size (minus 1 to leave space for
           a null terminator) bytes from the sender */
        if ((bytesRcvd = recv(sock, echoBuffer, RCVBUFSIZE - 1, 0)) <= 0)
            DieWithError("recv() failed or connection closed prematurely");
        totalBytesRcvd += bytesRcvd;   /* Keep tally of total bytes */
        echoBuffer[bytesRcvd] = '\0';  /* Terminate the string! */
        printf(echoBuffer);            /* Print the echo buffer */
    }

    printf("\n");    /* Print a final linefeed */

    close(sock);
    exit(0);
}

[root@lnxclient admin]# cat DieWithError.c
#include <stdio.h>  /* for perror() */
#include <stdlib.h> /* for exit() */

void DieWithError(char *errorMessage)
{
    perror(errorMessage);
    exit(1);
}

[root@lnxclie3nt admin]#
```

The steps in the program are fairly straightforward. First, we create a stream socket, and then establish the connection to the server. We send the string to echo, wait for the response, print it out, clean things up, and terminate. Now we can just compile the code and get ready to run it.

```
[root@lnxclient admin]# gcc –o TCPsocketClient TCPsocketClient.c DieWithError.c
[root@lnxclient admin]#
```

Before we run the program with **TCPsocketoClient** *<ServerIPAddress>* *<StringtoEcho>* *<ServerPort>*, we need to compile the server portion of the code on `lnxserver`. The code in these two files is more complex.

```
[root@lnxserver admin]# cat TCPsocketServer.c
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for socket(), bind(), and connect() */
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <string.h>     /* for memset() */
#include <unistd.h>     /* for close() */

#define MAXPENDING 5    /* Maximum outstanding connection requests */

void ErrorFunc(char *errorMessage);     /* Error handling function */
void HandleTCPClient(int clntSocket);   /* TCP client handling function */

int main(int argc, char *argv[])
{
    int servSock;                     /* Socket descriptor for server */
    int clntSock;                     /* Socket descriptor for client */
    struct sockaddr_in echoServAddr; /* Local address */
    struct sockaddr_in echoClntAddr; /* Client address */
    unsigned short echoServPort;     /* Server port */
    unsigned int clntLen;            /* Length of client address data
                                        structure */

    if (argc != 2)     /* Test for correct number of arguments */
    {
        fprintf(stderr, "Usage:  %s <Server Port>\n", argv[0]);
        exit(1);
    }

    echoServPort = atoi(argv[1]);  /* First arg:  local port */

    /* Create socket for incoming connections */
    if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        DieWithError("socket() failed");

    /* Construct local address structure */
    memset(&echoServAddr, 0, sizeof(echoServAddr));   /* Zero out
                                                         structure */
    echoServAddr.sin_family = AF_INET;                /* Internet address
                                                         family */
    echoServAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming
                                                         interface */
    echoServAddr.sin_port = htons(echoServPort);      /* Local port */

    /* Bind to the local address */


    if (bind(servSock, (struct sockaddr *) &echoServAddr,
        sizeof(echoServAddr)) < 0)
        DieWithError("bind() failed");

    /* Mark the socket so it will listen for incoming connections */
    if (listen(servSock, MAXPENDING) < 0)
        DieWithError("listen() failed");
    for (;;) /* Run forever */
    {
        /* Set the size of the in-out parameter */
        clntLen = sizeof(echoClntAddr);

        /* Wait for a client to connect */
        if ((clntSock = accept(servSock, (struct sockaddr *) &echoClntAddr,
                               &clntLen)) < 0)
            DieWithError("accept() failed");

        /* clntSock is connected to a client! */

        printf("Handling client %s\n", inet_ntoa(echoClntAddr.sin_addr));

        HandleTCPClient(clntSock);
    }
    /* NOT REACHED */
}

[root@lnxserver admin]# cat HandleTCPClient.c
#include <stdio.h>      /* for printf() and fprintf() */
#include <sys/socket.h> /* for recv() and send() */
#include <unistd.h>     /* for close() */

#define RCVBUFSIZE 32   /* Size of receive buffer */

void DieWithError(char *errorMessage);  /* Error handling function */

void HandleTCPClient(int clntSocket)
{
    char echoBuffer[RCVBUFSIZE];        /* Buffer for echo string */
    int recvMsgSize;                    /* Size of received message */

    /* Receive message from client */
    if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
        DieWithError("recv() failed");

    /* Send received string and receive again until end of transmission */
    while (recvMsgSize > 0)      /* zero indicates end of transmission */
    {
        /* Echo message back to client */
        if (send(clntSocket, echoBuffer, recvMsgSize, 0) != recvMsgSize)
            DieWithError("send() failed");

        /* See if there is more data to receive */
        if ((recvMsgSize = recv(clntSocket, echoBuffer, RCVBUFSIZE, 0)) < 0)
            DieWithError("recv() failed");
    }
    close(clntSocket);    /* Close client socket */
}

[root@lnxserver admin]#
```

The server socket performs a passive open and waits (forever, if need be) for the client to send a string for it to echo. It's the `HandleTCPClient.c` code that does the bulk of this work. We also need the `ErrorFunc.c` code, as before, so we have three files to compile instead of only two, as on the client side.

```
[root@lnxserver admin]# gcc -o TCPsocketServer TCPsocketServer.c
HandleTCPClient.c DieWithError.c
[root@lnxserver admin]#
```

Now we can start up the server on `lnxserver` using the syntax **TCPsocketServer** <*ServerPort*>. (Always check to make sure the port you choose is not in use already!)

```
[root@lnxserver admin]# ./TCPsocketServer 2005
```

The server just waits until the client on `lnxclient` makes a connection and presents a string for the server to echo. We'll use the string *TEST.*

```
[root@lnxclient admin]# ./TCPsocketClient 10.10.11.66 TEST 2005
Received: TEST
[root@lnxclient admin]#
```

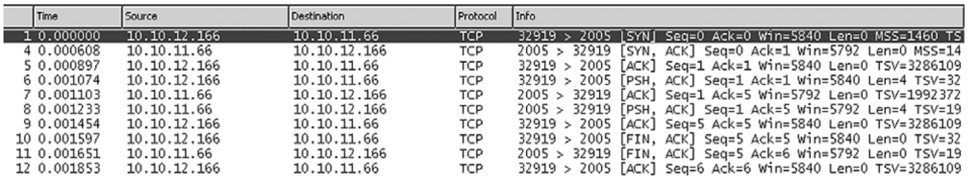
Not much to that. It's very fast, and the server tells us that the connection with `lnxclient` was made. We can cancel out of the server program.

```
Handling client 10.10.12.166
^C
[root@lnxserver admin]#
```

We've also used Ethereal to capture any TCP packets at the server while the socket client and server were running. Figure 12.4 shows what we caught.

So that's the attraction of sockets, especially for TCP. Ten packets (two ARPs are not shown) made their way back and forth across the network just to echo "TEST" from one system to another. Only two of the packets actually do this, as the rest are TCP connection overhead.

But the real power of sockets is in the details, or lack of details. Not a single line of C code mentioned creating a TCP or IP packet header, field values, or anything else. The stream socket interface did it all, so the application programmer can concentrate on the task at hand and not be forced to worry about network details.



```
   Time      Source          Destination     Protocol  Info
 1 0.000000  10.10.12.166    10.10.11.66     TCP       32919 > 2005 [SYN] Seq=0 Ack=0 Win=5840 Len=0 MSS=1460 TS
 4 0.000608  10.10.11.66     10.10.12.166    TCP       2005 > 32919 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=14
 5 0.000897  10.10.12.166    10.10.11.66     TCP       32919 > 2005 [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=3286109
 6 0.001074  10.10.12.166    10.10.11.66     TCP       32919 > 2005 [PSH, ACK] Seq=1 Ack=1 Win=5840 Len=4 TSV=32
 7 0.001103  10.10.11.66     10.10.12.166    TCP       2005 > 32919 [ACK] Seq=1 Ack=5 Win=5792 Len=0 TSV=1992372
 8 0.001233  10.10.11.66     10.10.12.166    TCP       2005 > 32919 [PSH, ACK] Seq=1 Ack=5 Win=5792 Len=4 TSV=19
 9 0.001454  10.10.12.166    10.10.11.66     TCP       32919 > 2005 [ACK] Seq=5 Ack=5 Win=5840 Len=0 TSV=3286109
10 0.001597  10.10.12.166    10.10.11.66     TCP       32919 > 2005 [FIN, ACK] Seq=5 Ack=5 Win=5840 Len=0 TSV=32
11 0.001651  10.10.11.66     10.10.12.166    TCP       2005 > 32919 [FIN, ACK] Seq=5 Ack=6 Win=5792 Len=0 TSV=19
12 0.001853  10.10.12.166    10.10.11.66     TCP       32919 > 2005 [ACK] Seq=6 Ack=6 Win=5840 Len=0 TSV=3286109
```

FIGURE 12.4

The socket client–server TCP stream captured. This is a completely normal TCP connection accomplished with a minimum of coded effort.

QUESTIONS FOR READERS

Figure 12.5 shows some of the concepts discussed in this chapter and can be used to help you answer the following questions.

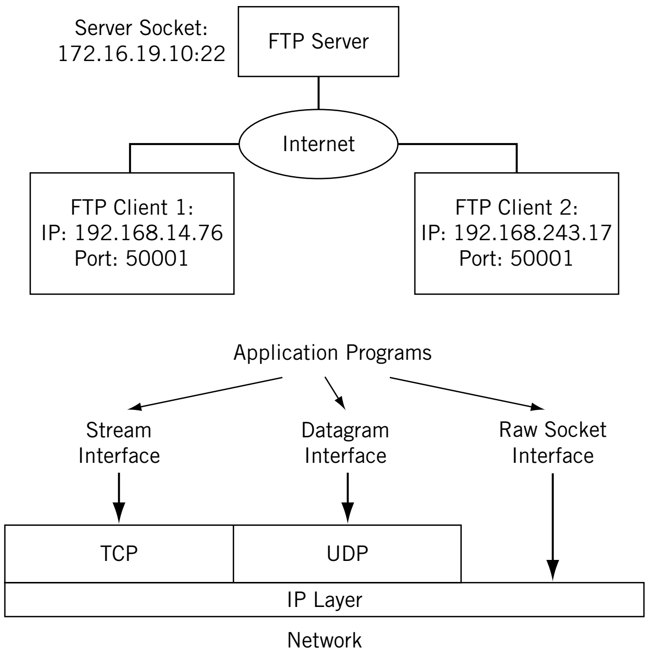


FIGURE 12.5

A socket in an FTP server and the various types of socket programming interfaces.

1. In the figure, two clients have picked the same ephemeral port for their FTP connection to the server. What is it about the TCP connection that allows this to happen all the time without harm?
2. What if the user at the *same* client PC ran two FTP sessions to the same server process? What would have to be different to make sure that both TCP control (and data) connections would not have problems?
3. What is the attraction of sockets as a programming tool?
4. Why can't the same type of socket interface be used for both TCP and UDP?
5. Are fully supported raw sockets an overstated threat to the Internet and attached hosts?