chapter 2

Basic TCP Sockets

It's time to learn about writing your own socket applications. We'll start with TCP. By now you're probably ready to get your hands dirty with some actual code, so we begin by going through a working example of a TCP client and server. Then we present the details of the socket API used in basic TCP. To keep things simpler, we'll present code initially that works for one particular version of IP: IPv4, which at the time this is being written is still the dominant version of the Internet Protocol, by a wide margin. At the end of this chapter we present the (minor) modifications required to write IPv6 versions of our clients and servers. In Chapter 3 we will demonstrate the creation of protocol-independent applications.

Our example client and server implement the *echo* protocol. It works as follows: the client connects to the server and sends its data. The server simply echoes whatever it receives back to the client and disconnects. In our application, the data that the client sends is a string provided as a command-line argument. Our client will print the data it receives from the server so we can see what comes back. Many systems include an echo service for debugging and testing purposes.

2.1 IPv4 TCP Client

The distinction between client and server is important because each uses the sockets interface differently at certain steps in the communication. We first focus on the client. Its job is to initiate communication with a server that is passively waiting to be contacted.

The typical TCP client's communication involves four basic steps:

1. Create a TCP socket using socket().
2. Establish a connection to the server using connect().
3. Communicate using send and recv().
4. Close the connection with close().

TCPEchoClient4.c is an implementation of a TCP echo client for IPv4.

TCPEchoClient4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "Practical.h"

int main(int argc, char *argv[]) {

  if (argc < 3 || argc > 4) // Test for correct number of arguments
    DieWithUserMessage("Parameter(s)",
        "<Server Address> <Echo Word> [<Server Port>]");

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

  // Third arg (optional): server port (numeric).  7 is well-known echo port
  in_port_t servPort = (argc == 4) ? atoi(argv[3]) : 7;

  // Create a reliable, stream socket using TCP
  int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (sock < 0)
    DieWithSystemMessage("socket() failed");

  // Construct the server address structure
  struct sockaddr_in servAddr;            // Server address
  memset(&servAddr, 0, sizeof(servAddr)); // Zero out structure
  servAddr.sin_family = AF_INET;          // IPv4 address family
  // Convert address
```

```c
  int rtnVal = inet_pton(AF_INET, servIP, &servAddr.sin_addr.s_addr);
  if (rtnVal == 0)
    DieWithUserMessage("inet_pton() failed", "invalid address string");
  else if (rtnVal < 0)
    DieWithSystemMessage("inet_pton() failed");
  servAddr.sin_port = htons(servPort);    // Server port

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

  size_t echoStringLen = strlen(echoString); // Determine input length

  // Send the string to the server
  ssize_t numBytes = send(sock, echoString, echoStringLen, 0);
  if (numBytes < 0)
    DieWithSystemMessage("send() failed");
  else if (numBytes != echoStringLen)
    DieWithUserMessage("send()", "sent unexpected number of bytes");

  // Receive the same string back from the server
  unsigned int totalBytesRcvd = 0; // Count of total bytes received
  fputs("Received: ", stdout);     // Setup to print the echoed string
  while (totalBytesRcvd < echoStringLen) {
    char buffer[BUFSIZE]; // I/O buffer
    /* Receive up to the buffer size (minus 1 to leave space for
     a null terminator) bytes from the sender */
    numBytes = recv(sock, buffer, BUFSIZE - 1, 0);
    if (numBytes < 0)
      DieWithSystemMessage("recv() failed");
    else if (numBytes == 0)
      DieWithUserMessage("recv()", "connection closed prematurely");
    totalBytesRcvd += numBytes; // Keep tally of total bytes
    buffer[numBytes] = '\0';    // Terminate the string!
    fputs(buffer, stdout);      // Print the echo buffer
  }

  fputc('\n', stdout); // Print a final linefeed

  close(sock);
  exit(0);
}
```

TCPEchoClient4.c

Our `TCPEchoClient4.c` does the following:

1. **Application setup and parameter parsing:** lines 1–21
   - **Include files:** lines 1–9
     These header files declare the standard functions and constants of the API. Consult your documentation (e.g., man pages) for the appropriate include files for socket functions and data structures on your system. We utilize our own include file, `Practical.h`, with prototypes for our own functions, which we describe below.
   - **Typical parameter parsing and sanity checking:** lines 13–21
     The IPv4 address and string to echo are passed in as the first two parameters. Optionally, the client takes the server port as the third parameter. If no port is provided, the client uses the well-known echo protocol port, 7.
2. **TCP socket creation:** lines 23–26
   We create a socket using the `socket()` function. The socket is for IPv4 (AF_INET) using the stream-based protocol (SOCK_STREAM) called TCP (IPPROTO_TCP). `socket()` returns an integer-valued descriptor or "handle" for the socket if successful. If `socket` fails, it returns –1, and we call our error-handling function, `DieWithSystemMessage()` (described later), to print an informative hint and exit.
3. **Prepare address and establish connection:** lines 28–42
   - **Prepare sockaddr_in structure to hold server address:** lines 29–30
     To connect a socket, we have to specify the address and port to connect to. The **sockaddr_in** structure is defined to be a "container" for this information. The call to `memset()` ensures that any parts of the structure that we do not explicitly set contain zero.
   - **Filling in the sockaddr_in:** lines 31–38
     We must set the address family (AF_INET), Internet address, and port number. The function `inet_pton()` converts the string representation of the server's Internet address (passed as a command-line argument in dotted-quad notation) into a 32-bit binary representation. The server's port number was converted from a command-line string to binary earlier; the call to `htons()` ("host to network short") ensures that the binary value is formatted as required by the API. (Reasons for this are described in Chapter 5.)
   - **Connecting:** lines 40–42
     The `connect()` function establishes a connection between the given socket and the one identified by the address and port in the **sockaddr_in** structure. Because the Sockets API is generic, the pointer to the **sockaddr_in** address structure (which is specific to IPv4 addresses) needs to be *cast* to the generic type (**sockaddr***), and the actual size of the address data structure must be supplied.
4. **Send echo string to server:** lines 44–51
   We find the length of the argument string and save it for later use. A pointer to the echo string is passed to the `send()` call; the string itself was stored somewhere (like all command-line arguments) when the application was started. We do not really care where

it is; we just need to know the address of the first byte and how many bytes to send. (Note that we do not send the end-of-string marker character (0) that is at the end of the argument string—and all strings in C). `send()` returns the number of bytes sent if successful and -1 otherwise. If `send()` fails or sends the wrong number of bytes, we must deal with the error. Note that sending the wrong number of bytes will not happen here. Nevertheless, it's a good idea to include the test because errors can occur in some contexts.

5. **Receive echo server reply:** lines 53–70
   TCP is a byte-stream protocol. One implication of this type of protocol is that `send()` boundaries are not preserved. In other words: **The bytes sent by a call to send() on one end of a connection may not all be returned by a single call to recv() on the other end.** (We discuss this issue in more detail in Chapter 7.) So we need to repeatedly receive bytes until we have received as many as we sent. In all likelihood, this loop will only be executed once because the data from the server will in fact be returned all at once; however, that is not *guaranteed* to happen, and so we have to allow for the possibility that multiple reads are required. This is a basic principle of writing applications that use sockets: **you must never assume anything about what the network and the program at the other end are going to do**.
   - **Receive a block of bytes:** lines 57–65
     `recv()` blocks until data is available, returning the number of bytes copied into the buffer or −1 in case of failure. A return value of zero indicates that the application at the other end closed the TCP connection. Note that the size parameter passed to `recv()` reserves space for adding a terminating null character.
   - **Print buffer:** lines 66–67
     We print the data sent by the server as it is received. We add the terminating null character (0) at the end of each chunk of received data so that it can be treated as a string by `fputs()`. We do not check whether the bytes received are the same as the bytes sent. The server may send something completely different (up to the length of the string we sent), and it will be written to the standard output.
   - **Print newline:** line 70
     When we have received as many bytes as we sent, we exit the loop and print a newline.
6. **Terminate connection and exit:** lines 72–73
   The `close()` function informs the remote socket that communication is ended, and then deallocates local resources of the socket.

Our client application (and indeed all the programs in this book) makes use of two error-handling functions:

```
DieWithUserMessage(const char *msg, const char *detail)
DieWithSystemMessage(const char *msg)
```

Both functions print a user-supplied message string (`msg`) to *stderr*, followed by a detail message string; they then call `exit()` with an error return code, causing the application to terminate.

The only difference is the source of the detail message. For DieWithUserMessage(), the detail message is user-supplied. For DieWithSystemMessage(), the detail message is supplied by the system based on the value of the special variable *errno* (which describes the reason for the most recent failure, if any, of a system call). We call DieWithSystemMessage() only if the error situation results from a call to a system call that sets *errno*. (To keep our programs simple, our examples do not contain much code devoted to recovering from errors—they simply punt and exit. Production code generally should not give up so easily.)

Occasionally, we need to supply information to the user without exiting; we use printf() if we need formatting capabilities, and fputs() otherwise. In particular, we try to avoid using printf() to output fixed, preformatted strings. **One thing that you should *never* do is to pass text received from the network as the first argument to printf(). It creates a serious security vulnerability. Use fputs() instead.**

> **Note:** the DieWith…() functions are *declared* in the header "Practical.h." However, the actual *implementation* of these functions is contained in the file DieWithMessage.c, which should be compiled and linked with *all* example applications in this text.

DieWithMessage.c

```
#include <stdio.h>
#include <stdlib.h>

void DieWithUserMessage(const char *msg, const char *detail) {
  fputs(msg, stderr);
  fputs(": ", stderr);
  fputs(detail, stderr);
  fputc('\n', stderr);
  exit(1);
}

void DieWithSystemMessage(const char *msg) {
  perror(msg);
  exit(1);
}
```

DieWithMessage.c

If we compile TCPEchoClient4.c and DieWithMessage.c to create program TCPEchoClient4, we can communicate with an echo server with Internet address 169.1.1.1 as follows:

```
% TCPEchoClient4 169.1.1.1 "Echo this!"
Received: Echo this!
```

For our client to work, we need a server. Many systems include an echo server for debugging and testing purposes; however, for security reasons, such servers are often initially disabled. If you don't have access to an echo server, that's okay because we're about to write one.

2.2 IPv4 TCP Server

We now turn our focus to constructing a TCP server. The server's job is to set up a communication endpoint and passively wait for a connection from the client. There are four general steps for basic TCP server communication:

1. Create a TCP socket using `socket()`.
2. Assign a port number to the socket with `bind()`.
3. Tell the system to allow connections to be made to that port, using `listen()`.
4. Repeatedly do the following:
   - Call `accept()` to get a new socket for each client connection.
   - Communicate with the client via that new socket using `send()` and `recv()`.
   - Close the client connection using `close()`.

Creating the socket, sending, receiving, and closing are the same as in the client. The differences in the server's use of sockets have to do with binding an address to the socket and then using the socket as a way to obtain other sockets that are connected to clients. (We'll elaborate on this in the comments following the code.) The server's communication with each client is as simple as can be: it simply receives data on the client connection and sends the same data back over to the client; it repeats this until the client closes its end of the connection, at which point no more data will be forthcoming.

TCPEchoServer4.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6  #include <netinet/in.h>
7  #include <arpa/inet.h>
8  #include "Practical.h"
9
10 static const int MAXPENDING = 5; // Maximum outstanding connection requests
11
12 int main(int argc, char *argv[]) {
  if (argc != 2) // Test for correct number of arguments
    DieWithUserMessage("Parameter(s)", "<Server Port>");

  in_port_t servPort = atoi(argv[1]); // First arg:  local port

  // Create socket for incoming connections
  int servSock; // Socket descriptor for server
  if ((servSock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
    DieWithSystemMessage("socket() failed");

  // Construct local address structure
  struct sockaddr_in servAddr;                  // Local address
  memset(&servAddr, 0, sizeof(servAddr));       // Zero out structure
  servAddr.sin_family = AF_INET;                // IPv4 address family
  servAddr.sin_addr.s_addr = htonl(INADDR_ANY); // Any incoming interface
  servAddr.sin_port = htons(servPort);          // Local port

  // Bind to the local address
  if (bind(servSock, (struct sockaddr*) &servAddr, sizeof(servAddr)) < 0)
    DieWithSystemMessage("bind() failed");

  // Mark the socket so it will listen for incoming connections
  if (listen(servSock, MAXPENDING) < 0)
    DieWithSystemMessage("listen() failed");

  for (;;) { // Run forever
    struct sockaddr_in clntAddr; // Client address
    // Set length of client address structure (in-out parameter)
    socklen_t clntAddrLen = sizeof(clntAddr);

    // Wait for a client to connect
    int clntSock = accept(servSock, (struct sockaddr *) &clntAddr, &clntAddrLen);
    if (clntSock < 0)
      DieWithSystemMessage("accept() failed");

    // clntSock is connected to a client!

    char clntName[INET_ADDRSTRLEN]; // String to contain client address
    if (inet_ntop(AF_INET, &clntAddr.sin_addr.s_addr, clntName,
        sizeof(clntName)) != NULL)
      printf("Handling client %s/%d\n", clntName, ntohs(clntAddr.sin_port));
    else
      puts("Unable to get client address");

```

```
58      HandleTCPClient(clntSock);
59    }
60    // NOT REACHED
61  }
```

TCPEchoServer4.c

1. **Program setup and parameter parsing:** lines 1–17
   We convert the port number from string to numeric value using `atoi()`; if the first argument is not a number, `atoi()` will return 0, which will cause an error later when we call `bind()`.
2. **Socket creation and setup:** lines 19–37
   - **Create a TCP socket:** lines 20–22
     We create a stream socket just like we did in the client.
   - **Fill in desired endpoint address:** lines 25–29
     On the server, we need to associate our server socket with an address and port number so that client connections get to the right place. Since we are writing for IPv4, we use a **sockaddr_in** structure for this. Because we don't much care which address we are on (any one assigned to the machine the server is running on will be OK), we let the system pick it by specifying the wildcard address INADDR_ANY as our desired Internet address. (This is usually the right thing to do for servers, and it saves the server from having to find out any actual Internet address.) Before setting both address and port number in the **sockaddr_in**, we convert each to network byte order using `htonl()` and `htons()`. (See Section 5.1.2 for details.)
   - **Bind socket to specified address and port:** lines 32–33
     As noted above, the server's socket needs to be associated with a local address and port; the function that accomplishes this is `bind()`. Notice that **while the client has to supply the server's address to `connect()`, the server has to specify its *own* address to `bind()`**. It is this piece of information (i.e., the server's address and port) that they have to agree on to communicate; neither one really needs to know the client's address. Note that `bind()` may fail for various reasons; one of the most important is that some other socket is already bound to the specified port (see Section 7.5). Also, on some systems special privileges are required to bind to certain ports (typically those with numbers less than 1024).
   - **Set the socket to listen:** lines 36–37
     The `listen()` call tells the TCP implementation to allow incoming connections from clients. Before the call to `listen()`, any incoming connection requests to the socket's address would be silently rejected—that is, the `connect()` would fail at the client.
3. **Iteratively handle incoming connections:** lines 39–59
   - **Accept an incoming connection:** lines 40–47

As discussed above, a TCP socket on which listen() has been called is used differently than the one we saw in the client application. Instead of sending and receiving on the socket, the server application calls accept(), which blocks until an incoming connection is made to the listening socket's port number. At that point, accept() returns a descriptor for a new socket, which is already connected to the initiating remote socket. The second argument points to a **sockaddr_in** structure, and the third argument is a pointer to the length of that structure. Upon success, the **sockaddr_in** contains the Internet address and port of the client to which the returned socket is connected; the address's length has been written into the integer pointed to by the third argument. Note that the socket referenced by the returned descriptor is already *connected*; among other things this means it is ready for sending and receiving. (For details about what happens in the underlying implementation, see Section 7.4.1 in Chapter 7.)

- **Report connected client:** lines 51–56
  At this point *clntAddr* contains the address and port number of the connecting client; we provide a "Caller ID" function and print out the client's information. As you might expect, inet_ntop() is the inverse of inet_pton(), which we used in the client. It takes the binary representation of the client's address and converts it to a dotted-quad string. Because the implementation deals with ports and addresses in so-called network byte order (Section 5.1.2), we have to convert the port number before passing it to printf() (inet_pton() takes care of this transparently for addresses).
- **Handle echo client:** line 58
  HandleTCPClient() takes care of the "application protocol." We discuss it below. Thus, we have factored out the "echo"-specific part of the server.

We have factored out the function that implements the "echo" part of our echo server. Although this *application protocol* only takes a few lines to implement, it's good design practice to isolate its details from the rest of the server code. This promotes code reuse.

HandleTCPClient() receives data on the given socket and sends it back on the same socket, iterating as long as recv() returns a positive value (indicating that something was received). recv() blocks until something is received or the client closes the connection. When the client closes the connection normally, recv() returns 0. You can find HandleTCPClient() in the file TCPServerUtility.c.

HandleTCPClient()

```
1  void HandleTCPClient(int clntSocket) {
2    char buffer[BUFSIZE]; // Buffer for echo string
3
4    // Receive message from client
5    ssize_t numBytesRcvd = recv(clntSocket, buffer, BUFSIZE, 0);
6    if (numBytesRcvd < 0)
7      DieWithSystemMessage("recv() failed");
```

```

  // Send received string and receive again until end of stream
  while (numBytesRcvd > 0) { // 0 indicates end of stream
    // Echo message back to client
    ssize_t numBytesSent = send(clntSocket, buffer, numBytesRcvd, 0);
    if (numBytesSent < 0)
      DieWithSystemMessage("send() failed");
    else if (numBytesSent != numBytesRcvd)
      DieWithUserMessage("send()", "sent unexpected number of bytes");

    // See if there is more data to receive
    numBytesRcvd = recv(clntSocket, buffer, BUFSIZE, 0);
    if (numBytesRcvd < 0)
      DieWithSystemMessage("recv() failed");
  }

  close(clntSocket); // Close client socket
}
```

HandleTCPClient()

Suppose we compile TCPEchoServer4.c, DieWithMessage.c, TCPServerUtility.c, and AddressUtility.c into the executable program TCPEchoServer4, and run that program on a host with Internet (IPv4) address 169.1.1.1, port 5000. Suppose also that we run our client on a host with Internet address 169.1.1.2 and connect it to the server. The server's output should look like this:

```
% TCPEchoServer4 5000
Handling client 169.1.1.2
```

While the client's output looks like this:

```
% TCPEchoClient4 169.1.1.1 "Echo this!" 5000
Received: Echo this!
```

The server binds its socket to port 5000 and waits for a connection request from the client. The client connects, sends the message "Echo this!" to the server, and receives the echoed response. In this command we have to supply TCPEchoClient with the port number on the command line because it is talking to our echo server, which is on port 5000 rather than the well-known port 7.

We have mentioned that a key principle for coding network applications using sockets is **Defensive Programming: your code must not make assumptions about anything received over the network**. What if you want to "play" with your TCP server to see how it responds to various incorrect client behaviors? You could write a TCP client that sends bogus messages and prints results; this, however, can be tedious and time-consuming. A quicker alternative is to use the **telnet** program available on most systems. This is a command-line tool that connects to a server, sends whatever text you type, and prints the response. Telnet takes two parameters: the server and port. For example, to telnet to our example echo server from above, try

```
% telnet 169.1.1.1 5000
```

Now type your string to echo and telnet will print the server response. The behavior of telnet differs between implementations, so you may need to research the specifics of its use on your system.

Now that we've seen a complete client and server, let's look at the individual functions that make up the Sockets API in a bit more detail.

2.3 Creating and Destroying Sockets

To communicate using TCP or UDP, a program begins by asking the operating system to create an instance of the socket abstraction. The function that accomplishes this is `socket()`; its parameters specify the flavor of socket needed by the program.

int `socket`(**int** *domain*, **int** *type*, **int** *protocol*)

The first parameter determines the communication *domain* of the socket. Recall that the Sockets API provides a generic interface for a large number of communication domains; however, we are only interested in IPv4 (AF_INET) and IPv6 (AF_INET6). Note that you may see some programs use PF_XXX here instead of AF_XXX. Typically, these values are equal, in which case they are interchangeable, but this is (alas) not guaranteed.[1]

The second parameter specifies the *type* of the socket. The type determines the semantics of data transmission with the socket—for example, whether transmission is reliable, whether message boundaries are preserved, and so on. The constant SOCK_STREAM specifies a socket with reliable byte-stream semantics, whereas SOCK_DGRAM specifies a best-effort datagram socket.

The third parameter specifies the particular *end-to-end protocol* to be used. For both IPv4 and IPv6, we want TCP (identified by the constant IPPROTO_TCP) for a stream socket, or UDP (identified by IPPROTO_UDP) for a datagram socket. Supplying the constant 0 as the third parameter causes the system to select the *default* end-to-end protocol for the specified protocol family and type. Because there is currently only one choice for stream sockets in the TCP/IP protocol family, we could specify 0 instead of giving the protocol number explicitly. Someday, however, there might be other end-to-end protocols in the Internet protocol family

[1]Truth be told, this is an ugly part of the Sockets interface, and the documentation is simply not helpful.

that implement the same semantics. In that case, specifying 0 might result in the use of a different protocol, which might or might not be desirable. The main thing is to ensure that the communicating programs are using the same end-to-end protocol.

We said earlier that `socket()` returns a *handle* for the communication instance. On UNIX-derived systems, it is an integer: a nonnegative value for success and −1 for failure. A nonfailure value should be treated as an opaque handle, like a file descriptor. (In reality, it *is* a file descriptor, taken from the same space as the numbers returned by `open()`.) This handle, which we call a *socket descriptor*, is passed to other API functions to identify the socket abstraction on which the operation is to be carried out.

When an application is finished with a socket, it calls `close()`, giving the descriptor for the socket that is no longer needed.

---

int `close`(**int** *`socket`*)

`close()` tells the underlying protocol stack to initiate any actions required to shut down communications and deallocate any resources associated with the socket. `close()` returns 0 on success or −1 on failure. Once `close()` has been called, invoking other operations (e.g., `send()` and `recv()`) on the socket results in an error.

2.4 Specifying Addresses

Applications using sockets need to be able to identify the remote endpoint(s) with which they will communicate. We've already seen that a client must specify the address and port number of the server application with which it needs to communicate. In addition, the sockets layer sometimes needs to pass addresses to the application. For example, a feature analogous to "Caller ID" in the telephone network lets a server know the address and port number of each client that communicates with it.

In this section, we describe the data structures used as containers for this information by the Sockets API.

2.4.1 Generic Addresses

The Sockets API defines a generic data type—the **sockaddr** structure—for specifying addresses associated with sockets:

```
struct sockaddr {
    sa_family_t sa_family;     // Address family (e.g., AF_INET)
    char sa_data[14];          // Family-specific address information
};
```

The first part of this address structure defines the address family—the space to which the address belongs. For our purposes, we will always use the system-defined constants AF_INET and AF_INET6, which specify the Internet address families for IPv4 and IPv6, respectively. The second part is a blob of bits whose exact form depends on the address family. (This is a typical way of dealing with heterogeneity in operating systems and networking.) As we discussed in Section 1.2, socket addresses for the Internet protocol family have two parts: a 32-bit (IPv4) or 128-bit (IPv6) Internet address and a 16-bit port number.[2]

2.4.2 IPv4 Addresses

The particular form of the **sockaddr** structure that is used for TCP/IP socket addresses depends on the IP version. For IPv4, use the **sockaddr_in** structure.

```
struct in_addr {
    uint32_t s_addr;       // Internet address (32 bits)
};

struct sockaddr_in {
    sa_family_t sin_family;    // Internet protocol (AF_INET)
    in_port_t sin_port;        // Address port (16 bits)
    struct in_addr sin_addr;   // IPv4 address (32 bits)
    char sin_zero[8];          // Not used
};
```

As you can see, the **sockaddr_in** structure has fields for the port number and Internet address in addition to the address family. It is important to understand that **sockaddr_in** is just a more detailed view of the data in a **sockaddr** structure, tailored to sockets using IPv4. Thus, we can fill in the fields of a **sockaddr_in** and then cast (a pointer to) it to a (pointer to a) **sockaddr** and pass it to the socket functions, which look at the sa_family field to learn the actual type, then cast back to the appropriate type.

2.4.3 IPv6 Addresses

For IPv6, use the **sockaddr_in6** structure.

```
struct in_addr {
    uint32_t s_addr[16];       // Internet address (128 bits)
};

struct sockaddr_in6 {
    sa_family_t sin6_family;   // Internet protocol (AF_INET6)
    in_port_t sin6_port;       // Address port (16 bits)
    uint32_t sin6_flowinfo;    // Flow information
```

[2]The astute reader may have noticed that the generic **sockaddr** structure is not big enough to hold both a 16-byte IPv6 address and a 2-byte port number. We'll deal with this difficulty shortly.

```
    struct in6_addr sin6_addr; // IPv6 address (128 bits)
    uint32_t sin6_scope_id;    // Scope identifier
};
```

The **sockaddr_in6** structure has additional fields beyond those of a **sockaddr_in**. These are intended for capabilities of the IPv6 protocol that are not commonly used. They will be (mostly) ignored in this book.

As with **sockaddr_in**, we must cast (a pointer to) the **sockaddr_in6** to (a pointer to) a **sockaddr** in order to pass it to the various socket functions. Again, the implementation uses the address family field to determine the actual type of the argument.

2.4.4 Generic Address Storage

If you know anything about how data structures are allocated in C, you may have already noticed that a **sockaddr** is not big enough to hold a **sockaddr_in6**. (If you don't know anything about it, don't fear: much of what you need to know will be covered in Chapter 5.) In particular, what if we want to allocate an address structure, but we don't know the actual address type (e.g., IPv4 or IPv6)? The generic **sockaddr** won't work because it's too small for some address structures.[3] To solve this problem, the socket designers created the **sockaddr_storage** structure, which is guaranteed to be as large as any supported address type.

```
struct sockaddr_storage {
    sa_family_t
    ...
    // Padding and fields to get correct length and alignment
    ...
};
```

As with **sockaddr**, we still have the leading family field to determine the actual type of the address; however, with **sockaddr_storage** we have sufficient space for any address type. (For a hint about how this could be accomplished, refer to the discussion of how the C compiler lays out structures in memory, in Section 5.1.6.)

One final note on addresses. On some platforms, the address structures contain an additional field that stores the length of the address structure in bytes. For **sockaddr**, **sockaddr_in**, **sockaddr_in6**, and **sockaddr_storage**, the extra fields are called `sa_len`, `sin_len`, `sin6_len`, and `ss_len`, respectively. Since a length field is not available on all systems, avoid using it. Typically, platforms that use this form of structure define a value (e.g., SIN6_LEN) that can be tested for at compile time to see if the length field is present.

[3]You may wonder why this is so (we do). The reasons apparently have to do with backward-compatibility: the Sockets API was first specified a long time ago, before IPv6, when resources were scarcer and there was no reason to have a bigger structure. Changing it now to make it bigger would apparently break binary-compatibility with some applications.

2.4.5 Binary/String Address Conversion

For socket functions to understand addresses, they must be in "numeric" (i.e., binary) form; however, addresses for human use are generally "printable" strings (e.g., 192.168.1.1 or 1::1). We can convert addresses from printable string to numeric using the `inet_pton()` function (**pton** = **p**rintable **to n**umeric):

int `inet_pton`(**int** *addressFamily*, **const char** **src*, **void** **dst*)

The first parameter, *addressFamily*, specifies the address family of the address being converted. Recall that the Sockets API provides a generic interface for a large number of communication domains. However, we are only interested here in IPv4 (AF_INET) and IPv6 (AF_INET6). The *src* parameter references a null-terminated character string containing the address to convert. The *dst* parameter points to a block of memory in the caller's space to hold the result; its length must be sufficient to hold the result (at least 4 bytes for IPv4 and 16 bytes for IPv6). `inet_pton()` returns 1 if the conversion succeeds, with the address referenced by *dst* in network byte order; 0 if the string pointed to by *src* is not formatted as a valid address; and −1 if the specified address family is unknown.

We can go the other way, converting addresses from numeric to printable form, using `inet_ntop()` (**ntop** = **n**umeric **to p**rintable):

const char *`inet_ntop`(**int** *addressFamily*, **const void** **src*, **char** **dst*, **socklen_t** *dstBytes*)

The first parameter, *addressFamily*, specifies the type of the address being converted. The second parameter *src* points to the first byte of a block of memory containing the numeric address to convert. The size of the block is determined by the address family. The *dst* parameter points to a buffer (block of memory) allocated in the caller's space, into which the resulting string will be copied; its size is given by *dstBytes*. How do we know what size to make the block of memory? The system-defined constants INET_ADDRSTRLEN (for IPv4) and INET6_ADDRSTRLEN (for IPv6) indicate the longest possible resulting string (in bytes). `inet_ntop()` returns a pointer to the string containing the printable address (i.e., the third argument) if the conversion succeeds and NULL otherwise.

2.4.6 Getting a Socket's Associated Addresses

The system associates a local and foreign address with each connected socket (TCP or UDP). Later we'll discuss the details of how these values are assigned. We can find out these addresses using `getsockname()` for the local address and `getpeername()` for the foreign address. Both methods return a **sockaddr** structure containing the Internet address and port information.

int getpeername(**int** *socket*, **struct sockaddr** **remoteAddress*, **socklen_t** **addressLength*)
int getsockname(**int** *socket*, **struct sockaddr** **localAddress*, **socklen_t** **addressLength*)

The *socket* parameter is the descriptor of the socket whose address information we want. The *remoteAddress* and *localAddress* parameters point to address structures into which the address information will be placed by the implementation; they are always cast to **sockaddr *** by the caller. **If we don't know the IP protocol version a priori, we should pass in a (pointer to a) sockaddr_storage to receive the result.** As with other socket calls using **sockaddr**, the *addressLength* is an in-out parameter specifying the length of the buffer (input) and returned address structure (output) in bytes.

2.5 Connecting a Socket

A TCP socket must be connected to another socket before any data can be sent through it. In this sense using TCP sockets is something like using the telephone network. Before you can talk, you have to specify the number you want, and a connection must be established; if the connection cannot be established, you have to try again later. The connection establishment process is the biggest difference between clients and servers: The client initiates the connection while the server waits passively for clients to connect to it. (For additional details about the connection process and how it relates to the API functions, see Section 7.4.) To establish a connection with a server, we call `connect()` on the socket.

int connect(**int** *socket*, **const struct sockaddr** **foreignAddress*, **socklen_t** *addressLength*)

The first argument, *socket*, is the descriptor created by `socket()`. *foreignAddress* is declared to be a pointer to a **sockaddr** because the Sockets API is generic; for our purposes, it will always be a pointer to either a **sockaddr_in** or **sockaddr_in6** containing the Internet address and port of the server. *addressLength* specifies the length of the address structure, typically given as `sizeof(struct sockaddr_in)` or `sizeof(struct sockaddr_in6)`. When `connect()` returns, the socket is connected, and communication can proceed with calls to `send()` and `recv()`.

2.6 Binding to an Address

As we have noted already, client and server "rendezvous" at the server's address and port. For that to work, the server must first be associated with that address and port. This is accomplished using `bind()`. Again, note that the client supplies the server's address to `connect()`, but the server has to specify its *own* address to `bind()`. Neither client nor server application needs to know the client's address in order for them to communicate. (Of course, the server may wish to know the client's address for logging or other purposes.)

int `bind`(**int** *`socket`*, **struct sockaddr** *`*localAddress`*, **socklen_t** *`addressSize`*)

The first parameter is the descriptor returned by an earlier call to `socket()`. As with `connect()`, the address parameter is declared as a pointer to a **sockaddr**, but for TCP/IP applications, it will always point to a **sockaddr_in** (for IPv4) or **sockaddr_in6** (for IPv6), containing the Internet address of the local interface and the port to listen on. The *`addressSize`* parameter is the size of the address structure. `bind()` returns 0 on success and −1 on failure.

It is important to realize that it is not possible for a program to bind a socket to an *arbitrary* Internet address—if a specific Internet address is given (of either type), the call will only succeed if that address is assigned to the host on which the program is running. A server on a host with multiple Internet addresses might bind to a specific one because it *only wants to accept connections that arrive to that address*. Typically, however, the server wants to accept connections sent to *any* of the host's addresses, and so sets the address part of the **sockaddr** to the "wildcard" address INADDR_ANY for IPv4 or *in6addr_any* for IPv6. The semantics of the wildcard address are that it matches any specific address. For a server, this means that it will receive connections addressed to any of the host's addresses (of the specified type).

While `bind()` is mostly used by servers, a client can also use `bind()` to specify its local address/port. For those TCP clients that don't pick their own local address/port with `bind()`, the local Internet address and port are determined during the call to `connect()`. Thus, a client must call `bind()` *before* calling `connect()` if it is going to use it.

You can initialize a **in6_addr** structure to the wildcard address with IN6ADDR_ANY_INIT; however, this special constant may only be used as an "initializer" in a declaration. **Note well that while INADDR_ANY is defined to be in host byte order and, consequently, must be converted to network byte order with htonl() before being used as an argument to `bind()`, *in6addr_any* and IN6ADDR_ANY_INIT are already in network byte order.**

Finally, if you supply the port number 0 to `bind()`, the system will select an unused local port for you.

2.7 Handling Incoming Connections

After binding, the server socket has an address (or at least a port). Another step is required to instruct the underlying protocol implementation to listen for connections from clients; this is done by calling `listen()` on the socket.

int listen(**int** *socket*, **int** *queueLimit*)

The listen() function causes internal state changes to the given socket, so that incoming TCP connection requests will be processed and then queued for acceptance by the program. (Section 7.4 in Chapter 7 has more details about the life cycle of a TCP connection.) The *queueLimit* parameter specifies an upper bound on the number of incoming connections that can be waiting at any time. The precise effect of *queueLimit* is very system dependent, so consult your local system's technical specifications.[4] listen() returns 0 on success and −1 on failure.

Once a socket is configured to listen, the program can begin accepting client connections on it. At first it might seem that a server should now wait for a connection on the socket that it has set up, send and receive through that socket, close it, and then repeat the process. However, that is not the way it works. The socket that has been bound to a port and marked "listening" is never actually used for sending and receiving. Instead, it is used as a way of getting *new* sockets, one for each client connection; the server then sends and receives on the *new* sockets. The server gets a socket for an incoming client connection by calling accept().

int accept(**int** *socket*, **struct sockaddr** **clientAddress*, **socklen_t** **addressLength*)

This function dequeues the next connection on the queue for *socket*. If the queue is empty, accept() blocks until a connection request arrives. When successful, accept() fills in the **sockaddr** structure pointed to by *clientAddress*, with the address and port of the client at the other end of the connection. Upon invocation, the *addressLength* parameter should specify the size of the structure pointed to by *clientAddress* (i.e., the space available); upon return it contains the size of the actual address returned. **A common beginner mistake is to fail to initialize the integer that *addressLength* points to so it contains the length of the structure that *clientAddress* points to**. The following shows the correct way:

```
struct sockaddr_storage address;
socklen_t addrLength = sizeof(address);
int newConnection = accept(sock, &address, &addrLength);
```

AQ1

If successful, accept() returns a descriptor for a *new* socket that is connected to the client. The socket passed as the first parameter to accept() is unchanged (not connected to the client) and continues to listen for new connection requests. On failure, accept() returns −1. On most systems, accept() only fails when passed a bad socket descriptior. However, on some platforms it may return an error if the new socket has experienced a network-level error after being created and before being accepted.

[4]For information about using "man" pages, see the preface.

2.8 Communication

Once a socket is "connected," you can begin sending and receiving data. As we've seen, a client creates a connected socket by calling `connect()`, and a connected socket is returned by `accept()` on a server. After connection, the distinction between client and server effectively disappears, at least as far as the Sockets API is concerned. Through a connected TCP socket, you can communicate using `send()` and `recv()`.

---

ssize_t `send`(**int** *socket*, **const void** **msg*, **size_t** *msgLength*, **int** *flags*)
ssize_t `recv`(**int** *socket*, **void** **rcvBuffer*, **size_t** *bufferLength*, **int** *flags*)

These functions have very similar arguments. The first parameter *socket* is the descriptor for the connected socket through which data is to be sent or received. For `send()`, *msg* points to the sequence of bytes to be sent, and *msgLength* is the number of bytes to send. The default behavior for `send()` is to block until all of the data is sent. (We revisit this behavior in Section 6.3 and Chapter 7.) For `recv()`, *rcvBuffer* points to the buffer—that is, an area in memory such as a character array—where received data will be placed, and *bufferLength* gives the length of the buffer, which is the maximum number of bytes that can be received at once. The default behavior for `recv()` is to block until at least some bytes can be transferred. (On most systems, the minimum amount of data that will cause the caller of `recv()` to unblock is 1 byte.)

The *flags* parameter in both `send()` and `recv()` provides a way to change some aspects of the default behavior of the socket call. Setting *flags* to 0 specifies the default behavior. `send()` and `recv()` return the number of bytes sent or received or −1 for failure. (See also Section 6.3.)

Remember: TCP is a byte-stream protocol, so `send()` boundaries are not preserved. **The number of bytes read in a single call to recv on the receiver is not necessarily determined by the number of bytes written by a single call to `send()`**. If you call `send()` with 3000 bytes, it may take several calls to `recv()` to get all 3000 bytes, even if you pass a 5000-byte buffer to each `recv()` call. If you call `send()` with 100 bytes four times, you might receive all 400 bytes with a single call to `recv()`. A common mistake when writing TCP socket applications involves assuming that if you write all of the data with one `send()` you can read it all with one `recv()`. All these possibilities are illustrated in Chapter 7.

2.9 Using IPv6

So far, we've seen a client and server that work only with IPv4. What if you want to use IPv6? The changes are relatively minor and basically involve using the IPv6 equivalents for the address structure and constants. Let's look at the IPv6 version of our TCP echo server.

TCPEchoServer6.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "Practical.h"

static const int MAXPENDING = 5; // Maximum outstanding connection requests

int main(int argc, char *argv[]) {

  if (argc != 2) // Test for correct number of arguments
    DieWithUserMessage("Parameter(s)", "<Server Port>");

  in_port_t servPort = atoi(argv[1]); // First arg:  local port

  // Create socket for incoming connections
  int servSock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
  if (servSock < 0)
    DieWithSystemMessage("socket() failed");

  // Construct local address structure
  struct sockaddr_in6 servAddr;           // Local address
  memset(&servAddr, 0, sizeof(servAddr)); // Zero out structure
  servAddr.sin6_family = AF_INET6;        // IPv6 address family
  servAddr.sin6_addr = in6addr_any;       // Any incoming interface
  servAddr.sin6_port = htons(servPort);   // Local port

  // Bind to the local address
  if (bind(servSock, (struct sockaddr *) &servAddr, sizeof(servAddr)) < 0)
    DieWithSystemMessage("bind() failed");

  // Mark the socket so it will listen for incoming connections
  if (listen(servSock, MAXPENDING) < 0)
    DieWithSystemMessage("listen() failed");

  for (;;) { // Run forever
    struct sockaddr_in6 clntAddr; // Client address
    // Set length of client address structure (in-out parameter)
    socklen_t clntAddrLen = sizeof(clntAddr);

44      // Wait for a client to connect
45      int clntSock = accept(servSock, (struct sockaddr *) &clntAddr, &clntAddrLen);
46      if (clntSock < 0)
47        DieWithSystemMessage("accept() failed");
48
49      // clntSock is connected to a client!
50
51      char clntName[INET6_ADDRSTRLEN]; // Array to contain client address string
52      if (inet_ntop(AF_INET6, &clntAddr.sin6_addr.s6_addr, clntName,
53          sizeof(clntName)) != NULL)
54        printf("Handling client %s/%d\n", clntName, ntohs(clntAddr.sin6_port));
55      else
56        puts("Unable to get client address");
57
58      HandleTCPClient(clntSock);
59    }
60    // NOT REACHED
61  }
```

TCPEchoServer6.c

1. **Socket creation:** lines 19–22
   We construct an IPv6 socket by specifying the communication domain as AF_INET6.
2. **Fill in local address:** lines 24–29
   For the local address, we use the IPv6 (**struct sockaddr_in6**) address structure and constants (AF_INET6 and *in6addr_any*). One subtle difference is that we do not have to convert *in6addr_any* to network byte order as we did with INADDR_ANY.
3. **Report connected client:** lines 51–56
   clntAddr, which contains the address of the connecting client, is declared as an IPv6 socket address structure. When we convert the numeric address representation to a string, the maximum string length is now INET6_ADDRSTRLEN. Finally, our call to `inet_ntop()` uses an IPv6 address.

You've now seen both IPv4- and IPv6-specific clients and servers. In Chapter 3 we will see how they can be made to work with either type of address.

Exercises

1. Experiment with the book's TCP echo server using telnet. What OS are you using? Does the server appear to echo as you type (character-by-character) or only after you complete a line?

2. Use telnet to connect to your favorite Web server on port 80 and fetch the default page. You can usually do this by sending the string "GET /" to the Web server. Report the server address/name and the text from the default page.
3. For `TCPEchoServer.c` we explicitly provide an address to the socket using `bind()`. We said that a socket must have an address for communication, yet we do not perform a `bind()` in `TCPEchoClient.c`. How is the echo client's socket given a local address?
4. Modify the client and server so that the server "talks" first, sending a greeting message, and the client waits until it has received the greeting before sending anything. What needs to be agreed upon between client and server?
5. Servers are supposed to run for a long time without stopping. Therefore, they have to be designed to provide good service no matter what their clients do. Examine the example `TCPEchoServer.c` and list anything you can think of that a client might do to cause the server to give poor service to other clients. Suggest improvements to fix the problems you find.
6. Using `getsockname()` and `getpeername()`, modify `TCPEchoClient4.c` to print the local and foreign address immediately after `connect()`.
7. What happens when you call `getpeername()` on an unconnected TCP socket?
8. Using `getsockname()` and `getpeername()`, modify `TCPEchoServer4.c` to print the local and foreign address for the server socket immediately before and after `bind()` and for the client socket immediately after it's returned by `accept()`.
9. Modify `TCPEchoClient4.c` to use **`bind()`** so that the system selects both the address and port.
10. Modify `TCPEchoClient4.c` so that the new version binds to a specific local address and system-selected port. If the local address changed or you moved the program to a host with a different local address, what do you think would happen?
11. What happens when you attempt to bind after calling **`connect()`**?
12. Why does the socket interface use a special socket to accept connections? In other words, what would be wrong with having a server create a socket, set it up using `bind()` and `listen()`, wait for a connection, send and receive through *that* socket, and then when it is finished, close it and repeat the process? (*Hint*: Think about what happens to connection requests that arrive right after the server closes the previous connection.)