chapter 4

Using UDP Sockets

The User Datagram Protocol (UDP) provides a simpler end-to-end service than TCP provides. In fact, UDP performs only two functions: (1) It adds another layer of addressing (ports) to that of IP; and (2) it detects data corruption that may occur in transit and discards any corrupted datagrams. Because of this simplicity, UDP (datagram) sockets have some different characteristics from the TCP (stream) sockets we saw earlier.

For example, UDP sockets do not have to be connected before being used. Where TCP is analogous to telephone communication, UDP is analogous to communicating by mail: You do not have to "connect" before you send a package or letter, but you do have to specify the destination address for each one. In receiving, a UDP socket is like a mailbox into which letters or packages from many different sources can be placed.

Another difference between UDP sockets and TCP sockets is the way they deal with message boundaries: UDP sockets preserve them. This makes receiving an application message simpler, in some ways, than with TCP sockets. We will discuss this further in Section 4.3. A final difference is that the end-to-end transport service UDP provides is best effort: There is no guarantee that a message sent via a UDP socket will arrive at its destination. This means that a program using UDP sockets must be prepared to deal with loss and reordering of messages; we'll see an example of that later.

Again we introduce the UDP portion of the Sockets API through simple client and server programs. As before, they implement a trivial echo protocol. Afterward, we describe the API functionality in more detail in Sections 4.3 and 4.4.

4.1 UDP Client

Our UDP echo client, `UDPEchoClient.c`, looks similar to our address-family-independent `TCPEchoClient.c` in the way it sets up the server address and communicates with the server. However, it does not call `connect()`; it uses `sendto()` and `recvfrom()` instead of `send()` and `recv()`; and it only needs to do a single receive because UDP sockets preserve message boundaries, unlike TCP's byte-stream service. Of course, a UDP client only communicates with a UDP server. Many systems include a UDP echo server for debugging and testing purposes; the server simply echoes whatever messages it receives back to wherever they came from. After setting up, our echo client performs the following steps: (1) it sends the echo string to the server, (2) it receives the echo, and (3) it shuts down the program.

UDPEchoClient.c

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <string.h>
 4  #include <unistd.h>
 5  #include <sys/socket.h>
 6  #include <netdb.h>
 7  #include "Practical.h"
 8
 9  int main(int argc, char *argv[]) {
10
11    if (argc < 3 || argc > 4) // Test for correct number of arguments
12      DieWithUserMessage("Parameter(s)",
13          "<Server Address/Name> <Echo Word> [<Server Port/Service>]");
14
15    char *server = argv[1];     // First arg: server address/name
16    char *echoString = argv[2]; // Second arg: word to echo
17
18    size_t echoStringLen = strlen(echoString);
19    if (echoStringLen > MAXSTRINGLENGTH) // Check input length
20      DieWithUserMessage(echoString, "string too long");
21
22    // Third arg (optional): server port/service
23    char *servPort = (argc == 4) ? argv[3] : "echo";
24
25    // Tell the system what kind(s) of address info we want
26    struct addrinfo addrCriteria;                   // Criteria for address match
27    memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
28    addrCriteria.ai_family = AF_UNSPEC;             // Any address family
29    // For the following fields, a zero value means "don't care"
```

```
30    addrCriteria.ai_socktype = SOCK_DGRAM;          // Only datagram sockets
31    addrCriteria.ai_protocol = IPPROTO_UDP;         // Only UDP protocol
32
33    // Get address(es)
34    struct addrinfo *servAddr; // List of server addresses
35    int rtnVal = getaddrinfo(server, servPort, &addrCriteria, &servAddr);
36    if (rtnVal != 0)
37      DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
38
39    // Create a datagram/UDP socket
40    int sock = socket(servAddr->ai_family, servAddr->ai_socktype,
41        servAddr->ai_protocol); // Socket descriptor for client
42    if (sock < 0)
43      DieWithSystemMessage("socket() failed");
44
45    // Send the string to the server
46    ssize_t numBytes = sendto(sock, echoString, echoStringLen, 0,
47        servAddr->ai_addr, servAddr->ai_addrlen);
48    if (numBytes < 0)
49      DieWithSystemMessage("sendto() failed");
50    else if (numBytes != echoStringLen)
51      DieWithUserMessage("sendto() error", "sent unexpected number of bytes");
52
53    // Receive a response
54
55    struct sockaddr_storage fromAddr; // Source address of server
56    // Set length of from address structure (in-out parameter)
57    socklen_t fromAddrLen = sizeof(fromAddr);
58    char buffer[MAXSTRINGLENGTH + 1]; // I/O buffer
59    numBytes = recvfrom(sock, buffer, MAXSTRINGLENGTH, 0,
60        (struct sockaddr *) &fromAddr, &fromAddrLen);
61    if (numBytes < 0)
62      DieWithSystemMessage("recvfrom() failed");
63    else if (numBytes != echoStringLen)
64      DieWithUserMessage("recvfrom() error", "received unexpected number of bytes");
65
66    // Verify reception from expected source
67    if (!SockAddrsEqual(servAddr->ai_addr, (struct sockaddr *) &fromAddr))
68      DieWithUserMessage("recvfrom()", "received a packet from unknown source");
69
70    freeaddrinfo(servAddr);
71
72    buffer[echoStringLen] = '\0';     // Null-terminate received data
73    printf("Received: %s\n", buffer); // Print the echoed string
74
```

```
75    close(sock);
76    exit(0);
77  }
```

UDPEchoClient.c

1. **Program setup and parameter parsing:** lines 1–23
   The server address/name and string to echo are passed in as the first two parameters. We restrict the size of our echo message; therefore, we must verify that the given string satisfies this restriction. Optionally, the client takes the server port or service name as the third parameter. If no port is provided, the client uses the well-known echo protocol service name, "echo".
2. **Get foreign address for server:** lines 25–37
   For the server, we may be given an IPv4 address, IPv6 address, or name to resolve. For the optional port, we may be given a port number or service name. We use `getaddrinfo()` to determine the corresponding address information (i.e., family, address, and port number). Note that we'll accept an address for any family (AF_UNSPEC) for UDP (SOCK_DGRAM and IPPROTO_UDP); specifying the latter two effectively restricts the returned address families to IPv4 and IPv6. Note also that `getaddrinfo()` may return multiple addresses; by simply using the first, we may fail to communicate with the server when communication is possible using an address later in the list. **A production client should be prepared to try all returned addresses.**

3. **Socket creation and setup:** lines 39–43
   This is almost identical to the TCP echo client, except that we create a datagram socket using UDP. Note that we do not need to `connect()` before communicating with the server.
4. **Send a single echo datagram:** lines 45–51
   With UDP we simply tell `sendto()` the datagram destination. If we wanted to, we could call `sendto()` multiple times, changing the destination on every call, thus communicating with multiple servers through the same socket. The first call to `sendto()` also assigns an arbitrarily chosen local port number, not in use by any other socket, to the socket identified by *sock*, because we have not previously bound the socket to a port number. We do not know (or care) what the chosen port number is, but the server will use it to send the echoed message back to us.
5. **Get and print echo reply:** lines 55–73
   - **Receive a message:** lines 55–64
     We initialize *fromAddrLen* to contain the size of the address buffer (*fromAddr*) and then pass its address as the last parameter. `recvfrom()` blocks until a UDP datagram addressed to this socket's port arrives. It then copies the data from the first arriving datagram into *buffer* and copies the Internet address and (UDP) port number of its source from the packet's headers into the structure *fromAddr*. Note that the data buffer

is actually one byte bigger than MAXSTRINGLENGTH, which allows us to add a null byte to terminate the string.

- **Check message source:** lines 67–70
  Because there is no connection, a received message can come from any source. The output parameter *fromAddr* informs us of the datagram's source, and we check it to make sure it matches the server's Internet address. We use our own function, `SockAddrsEqual()`, to perform protocol-independent comparison of socket addresses. Although it is very unlikely that a packet would ever arrive from any other source, we include this check to emphasize that it is possible. There's one other complication. For applications with multiple or repeated requests, we must keep in mind that UDP messages may be reordered and arbitrarily delayed, so simply checking the source address and port may not be sufficient. (For example, the DNS protocol over UDP uses an identifier field to link requests and responses and detect duplication.) This is our last use of the address returned from `getaddrinfo()`, so we can free the associated storage.
- **Print received string:** lines 72–73
  Before printing the received data as a string, we first ensure that it is null-terminated.

6. **Wrap-up:** lines 75–76

This example client is fine as an introduction to the UDP socket calls; it will work correctly most of the time. However, **it would not be suitable for production use, because if a message is lost going to or from the server, the call to recvfrom() blocks forever, and the program does not terminate.** Clients generally deal with this problem through the use of *timeouts*, a subject we cover later, in Section 6.3.3.

4.2 UDP Server

Our next example program implements the UDP version of the echo server, `UDPEchoServer.c`. The server is very simple: It loops forever, receiving a message and then sending the same message back to wherever it came from. Actually, the server only receives and sends back the first 255 characters of the message; any excess is silently discarded by the sockets implementation. (See Section 4.3 for an explanation.)

UDPEchoServer.c

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <sys/types.h>
4  #include <sys/socket.h>
5  #include <netdb.h>
6  #include "Practical.h"
```

```
 7
 8  int main(int argc, char *argv[]) {
 9
10    if (argc != 2) // Test for correct number of arguments
11      DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
12
13    char *service = argv[1]; // First arg:  local port/service
14
15    // Construct the server address structure
16    struct addrinfo addrCriteria;                   // Criteria for address
17    memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
18    addrCriteria.ai_family = AF_UNSPEC;             // Any address family
19    addrCriteria.ai_flags = AI_PASSIVE;             // Accept on any address/port
20    addrCriteria.ai_socktype = SOCK_DGRAM;          // Only datagram socket
21    addrCriteria.ai_protocol = IPPROTO_UDP;         // Only UDP socket
22
23    struct addrinfo *servAddr; // List of server addresses
24    int rtnVal = getaddrinfo(NULL, service, &addrCriteria, &servAddr);
25    if (rtnVal != 0)
26      DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
27
28    // Create socket for incoming connections
29    int sock = socket(servAddr->ai_family, servAddr->ai_socktype,
30        servAddr->ai_protocol);
31    if (sock < 0)
32      DieWithSystemMessage("socket() failed");
33
34    // Bind to the local address
35    if (bind(sock, servAddr->ai_addr, servAddr->ai_addrlen) < 0)
36      DieWithSystemMessage("bind() failed");
37
38    // Free address list allocated by getaddrinfo()
39    freeaddrinfo(servAddr);
40
41    for (;;) { // Run forever
42      struct sockaddr_storage clntAddr; // Client address
43      // Set Length of client address structure (in-out parameter)
44      socklen_t clntAddrLen = sizeof(clntAddr);
45
46      // Block until receive message from a client
47      char buffer[MAXSTRINGLENGTH]; // I/O buffer
48      // Size of received message
49      ssize_t numBytesRcvd = recvfrom(sock, buffer, MAXSTRINGLENGTH, 0,
50          (struct sockaddr *) &clntAddr, &clntAddrLen);
51      if (numBytesRcvd < 0)
```

```
52        DieWithSystemMessage("recvfrom() failed");
53
54      fputs("Handling client ", stdout);
55      PrintSocketAddress((struct sockaddr *) &clntAddr, stdout);
56      fputc('\n', stdout);
57
58      // Send received datagram back to the client
59      ssize_t numBytesSent = sendto(sock, buffer, numBytesRcvd, 0,
60          (struct sockaddr *) &clntAddr, sizeof(clntAddr));
61      if (numBytesSent < 0)
62        DieWithSystemMessage("sendto() failed)");
63      else if (numBytesSent != numBytesRcvd)
64        DieWithUserMessage("sendto()", "sent unexpected number of bytes");
65    }
66    // NOT REACHED
67  }
```

UDPEchoServer.c

1. **Program setup and parameter parsing:** lines 1–13
2. **Parse/resolve address/port args:** lines 15–26
   The port may be specified on the command line as a port number or service name. We use `getaddrinfo()` to determine the actual local port number. As with our UDP client, we'll accept an address for any family (AF_UNSPEC) for UDP (SOCK_DGRAM and IPPROTO_UDP). We want our UDP server to accept echo requests from any of its interfaces. Setting the AI_PASSIVE flag makes `getaddrinfo()` return the wildcard Internet address (INADDR_ANY for IPv4 or *in6addr_any* for IPv6). `getaddrinfo()` may return multiple addresses; we simply use the first.
3. **Socket creation and setup:** lines 28–39
   This is nearly identical to the TCP echo server, except that we create a datagram socket using UDP. Also, we do not need to call `listen()` because there is no connection setup—the socket is ready to receive messages as soon as it has an address.
4. **Iteratively handle incoming echo requests:** lines 41–65
   Several key differences between UDP and TCP servers are demonstrated in how each communicates with the client. In the TCP server, we blocked on a call to `accept()` awaiting a connection from a client. Since UDP servers do not establish a connection, we do not need to get a new socket for each client. Instead, we can immediately call `recvfrom()` with the same socket that was bound to the desired port number.
   - **Receive an echo request:** lines 42–52
     `recvfrom()` blocks until a datagram is received from a client. Since there is no connection, each datagram may come from a different sender, and we learn the source at

the same time we receive the datagram. `recvfrom()` puts the address of the source in *clntAddr*. The length of this address buffer is specified by *cliAddrLen*.

- **Send echo reply:** lines 59–64
  `sendto()` transmits the data in *buffer* back to the address specified by *clntAddr*. Each received datagram is considered a single client echo request, so we only need a single send and receive—unlike the TCP echo server, where we needed to receive until the client closed the connection.

4.3 Sending and Receiving with UDP Sockets

As soon as it is created, a UDP socket can be used to send/receive messages to/from any address and to/from many *different* addresses in succession. To allow the destination address to be specified for each message, the Sockets API provides a different sending routine that is generally used with UDP sockets: `sendto()`. Similarly, the `recvfrom()` routine returns the source address of each received message in addition to the message itself.

---

ssize_t `sendto`(**int** *socket*, **const void** **msg*, **size_t** *msgLength*, **int** *flags*,
const struct sockaddr **destAddr*, **socklen_t** *addrLen*)
ssize_t `recvfrom`(**int** *socket*, **void** **msg*, **size_t** *msgLength*, **int** *flags*,
struct sockaddr **srcAddr*, **socklen_t** **addrLen*)

The first four parameters to `sendto()` are the same as those for `send()`. The two additional parameters specify the message's destination. Again, they will invariably be a pointer to a **struct sockaddr_in** and its size, respectively, or a pointer to a **struct sockaddr_in6** and its size, respectively. Similarly, `recvfrom()` takes the same parameters as `recv()` but, in addition, has two parameters that inform the caller of the source of the received datagram. One thing to note is that *addrLen* is an *in-out* parameter in `recvfrom()`: On input it specifies the size of the address buffer *srcAddr*, which will typically be a **struct sockaddr_storage** in IP-version-independent code. On output, it specifies the size of the address that was actually copied into the buffer. **Two errors often made by novices are (1) passing an integer value instead of a pointer to an integer for *addrLen* and (2) forgetting to initialize the pointed-to length variable to contain the appropriate size.**

We have already pointed out a subtle but important difference between TCP and UDP, namely, that *UDP preserves message boundaries*. In particular, each call to `recvfrom()` returns data from at most one `sendto()` call. Moreover, different calls to `recvfrom()` will never return data from the same call to `sendto()` (unless you use the MSG_PEEK flag with `recvfrom()`—see the last paragraph of this section).

When a call to `send()` on a TCP socket returns, all the caller knows is that the data has been copied into a buffer for transmission; the data may or may not have actually been transmitted

yet. (This is explained in more detail in Chapter 7.) However, UDP does not buffer data for possible retransmission because it does not recover from errors. This means that by the time a call to `sendto()` on a UDP socket returns, the message has been passed to the underlying channel for transmission and is (or soon will be) on its way out the door.

Between the time a message arrives from the network and the time its data is returned via `recv()` or `recvfrom()`, the data is stored in a first-in, first-out (FIFO) receive buffer. With a connected TCP socket, all received-but-not-yet-delivered bytes are treated as one continuous sequence (see Section 7.1). For a UDP socket, however, the bytes from different messages may have come from different senders. Therefore, the boundaries between them need to be preserved so that the data from each message can be returned with the proper address. The buffer really contains a FIFO sequence of "chunks" of data, each with an associated source address. A call to `recvfrom()` will never return more than one of these chunks. However, if `recvfrom()` is called with size parameter *n*, and the size of the first chunk in the receive FIFO is bigger than *n*, only the first *n* bytes of the chunk are returned. **The remaining bytes are quietly discarded, with no indication to the receiving program**.

For this reason, a receiver should always supply a buffer big enough to hold the largest message allowed by its application protocol at the time it calls `recvfrom()`. This technique will guarantee that no data will be lost. The maximum amount of data that can ever be returned by `recvfrom()` on a UDP socket is 65,507 bytes—the largest payload that can be carried in a UDP datagram.

Alternatively, the receiver can use the MSG_PEEK flag with `recvfrom()` to "peek" at the first chunk waiting to be received. This flag causes the received data to remain in the socket's receive FIFO so it can be received more than once. This strategy can be useful if memory is scarce, application messages vary widely in size, and each message carries information about its size in the first few bytes. The receiver first calls `recvfrom()` with MSG_PEEK and a small buffer, examines the first few bytes of the message to determine its size, and then calls `recvfrom()` again (without MSG_PEEK) with a buffer big enough to hold the entire message. In the usual case where memory is not scarce, using a buffer big enough for the largest possible message is simpler.

4.4 Connecting a UDP Socket

It is possible to call `connect()` on a UDP socket to fix the destination address of future datagrams sent over the socket. Once connected, you may use `send()` instead of `sendto()` to transmit datagrams because you no longer need to specify the destination address. In a similar way, you may use `recv()` instead of `recvfrom()` because a connected UDP socket can *only* receive datagrams from the associated foreign address and port, so after calling `connect()` you know the source address of any incoming datagrams. In fact, after connecting, you may *only* send and receive to/from the address specified to `connect()`. Note that connecting and then using `send()` and `recv()` with UDP does not change how UDP behaves. Message boundaries are still

preserved, datagrams can be lost, and so on. You can "disconnect" by calling `connect()` with an address family of AF_UNSPEC.

Another subtle advantage to calling `connect()` on a UDP socket is that it enables you to receive error indications that result from earlier actions on the socket. The canonical example is sending a datagram to a nonexistent server or port. When this happens, the `send()` that eventually leads to the error returns with no indication of error. Some time later, an error message is delivered to your host, indicating that the sent datagram encountered a problem. Because this datagram is a *control* message and not a regular UDP datagram, the system can't always tell where to send it if your socket is unconnected, because an unconnected socket has no associated foreign address and port. However, if your socket is connected, the system is able to match the information in the error datagram with your socket's associated foreign IP address and port. (See Section 7.5 for details about this process.) Note such a control error message being delivered to your socket will result in an error return from a *subsequent* system call (for example, the `recv()` that was intended to get the reply), not the offending `send()`.

Exercises

1. Modify `UDPEchoClient.c` to use `connect()`. After the final `recv()`, show how to disconnect the UDP socket. Using `getsockname()` and `getpeername()`, print the local and foreign address before and after `connect()`, and after disconnect.
2. Modify `UDPEchoServer.c` to use `connect()`.
3. Verify experimentally the size of the largest datagram you can send and receive using a UDP socket. Is the answer different for IPv4 and IPv6?
4. While `UDPEchoServer.c` explicitly specifies its local port number using `bind()`, we do not call `bind()` in `UDPEchoClient.c`. How is the UDP echo client's socket given a port number? Note that the answer is different for UDP and TCP. We can select the client's local port using `bind()`. What difficulties might we encounter if we do this?
5. Modify `UDPEchoClient.c` and `UDPEchoServer.c` to allow the largest echo string possible where an echo request is restricted to a single datagram.
6. Modify `UDPEchoClient.c` and `UDPEchoServer.c` to allow arbitrarily large echo strings. You may ignore datagram loss and reordering (for now).
7. Using `getsockname()` and `getpeername()`, modify `UDPEchoClient.c` to print the local and foreign address for the socket immediately before and after `sendto()`.
8. You can use the same UDP socket to send datagrams to many different destinations. Modify `UDPEchoClient.c` to send and receive an echo datagram to/from two different UDP echo servers. You can use the book's server running on multiple hosts or twice on the same host with different ports.