## 8.1 Introduction

There are some fundamental differences between applications written using TCP versus those that use UDP. These are because of the differences in the two transport layers: UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream provided by TCP. Nevertheless, there are instances when it makes sense to use UDP instead of TCP, and we will go over this design choice in Section 22.4. Some popular applications are built using UDP: DNS, NFS, and SNMP, for example.

Figure 8.1 shows the function calls for a typical UDP client/server. The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the `sendto` function (described in the next section), which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the `recvfrom` function, which waits until data arrives from some client. `recvfrom` returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

**Figure 8.1. Socket functions for UDP client/server.**

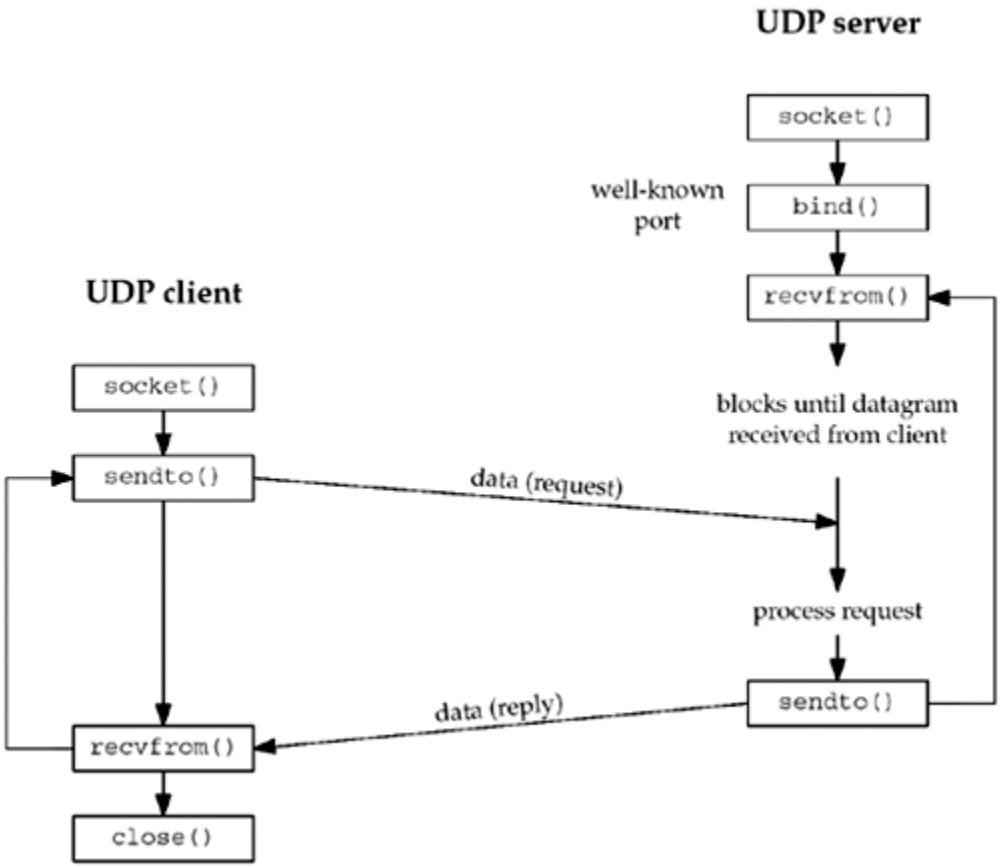



Figure 8.1 shows a timeline of the typical scenario that takes place for a UDP client/server exchange. We can compare this to the typical TCP exchange that was shown in Figure 4.1.

In this chapter, we will describe the new functions that we use with UDP sockets, `recvfrom` and `sendto`, and redo our echo client/server to use UDP. We will also describe the use of the `connect` function with a UDP socket, and the concept of asynchronous errors.

## 8.2 recvfrom and sendto Functions

These two functions are similar to the standard read and write functions, but three additional arguments are required.

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr *from, socklen_t *addrlen);

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct sockaddr *to, socklen_t addrlen);
```

Both return: number of bytes read or written if OK, −1 on error

The first three arguments, *sockfd, buff*, and *nbytes*, are identical to the first three arguments for read and write: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.

We will describe the *flags* argument in Chapter 14 when we discuss the recv, send, recvmsg, and sendmsg functions, since we do not need them with our simple UDP client/server example in this chapter. For now, we will always set the *flags* to 0.

The *to* argument for sendto is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent. The size of this socket address structure is specified by *addrlen*. The recvfrom function fills in the socket address structure pointed to by *from* with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by *addrlen*. Note that the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value-result argument).

The final two arguments to recvfrom are similar to the final two arguments to accept: The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP). The final two arguments to sendto are similar to the final two arguments to connect: We fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).

Both functions return the length of the data that was read or written as the value of the function. In the typical use of recvfrom, with a datagram protocol, the return value is the amount of user data in the datagram received.

Writing a datagram of length 0 is acceptable. In the case of UDP, this results in an IP datagram containing an IP header (normally 20 bytes for IPv4 and 40 bytes for IPv6), an 8-byte UDP header, and no data. This also means that a return value of 0 from recvfrom is acceptable for a datagram protocol: It does not mean that the peer has closed the connection, as does a return value of 0 from read on a TCP socket. Since UDP is connectionless, there is no such thing as closing a UDP connection.

If the *from* argument to recvfrom is a null pointer, then the corresponding length argument (*addrlen*) must also be a null pointer, and this indicates that we are not interested in knowing the protocol address of who sent us data.

Both recvfrom and sendto can be used with TCP, although there is normally no reason to do so.

## 8.3 UDP Echo Server: `main` Function

We will now redo our simple echo client/server from Chapter 5 using UDP. Our UDP client and server programs follow the function call flow that we diagrammed in Figure 8.1. Figure 8.2 depicts the functions that are used. Figure 8.3 shows the server `main` function.

**Figure 8.2. Simple echo client/server using UDP.**

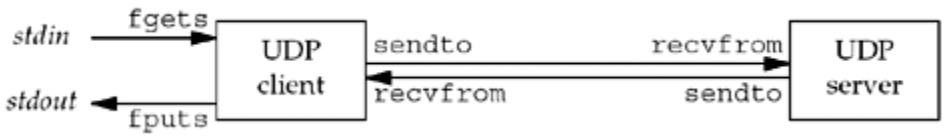



**Figure 8.3 UDP echo server.**

*udpcliserv/udpserv01.c*

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_in servaddr, cliaddr;

 7     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

 8     bzero(&servaddr, sizeof(servaddr));
 9     servaddr.sin_family = AF_INET;
10     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11     servaddr.sin_port = htons(SERV_PORT);

12     Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

13     dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14 }
```

### Create UDP socket, bind server's well-known port

*7–12* We create a UDP socket by specifying the second argument to `socket` as `SOCK_DGRAM` (a datagram socket in the IPv4 protocol). As with the TCP server example, the IPv4 address for the `bind` is specified as `INADDR_ANY` and the server's well-known port is the constant `SERV_PORT` from the `unp.h` header.

*13* The function `dg_echo` is called to perform server processing.

# 8.4 UDP Echo Server: dg_echo Function

Figure 8.4 shows the dg_echo function.

### Figure 8.4 dg_echo function: echo lines on a datagram socket.

*lib/dg_echo.c*

```
 1 #include     "unp.h"

 2 void
 3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 4 {
 5     int     n;
 6     socklen_t len;
 7     char    mesg[MAXLINE];

 8     for ( ; ; ) {
 9         len = clilen;
10         n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

11         Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12     }
13 }
```

## Read datagram, echo back to sender

*8–12* This function is a simple loop that reads the next datagram arriving at the server's port using recvfrom and sends it back using sendto.

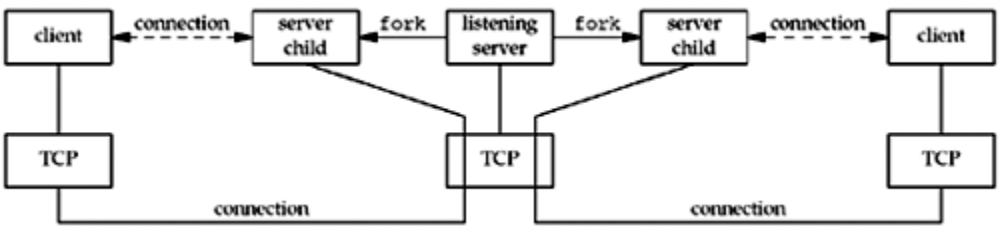
Despite the simplicity of this function, there are numerous details to consider. First, this function never terminates. Since UDP is a connectionless protocol, there is nothing like an EOF as we have with TCP.

Next, this function provides an *iterative server*, not a concurrent server as we had with TCP. There is no call to fork, so a single server process handles any and all clients. In general, most TCP servers are concurrent and most UDP servers are iterative.

There is implied queuing taking place in the UDP layer for this socket. Indeed, each UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer. When the process calls recvfrom, the next datagram from the buffer is returned to the process in a first-in, first-out (FIFO) order. This way, if multiple datagrams arrive for the socket before the process can read what's already queued for the socket, the arriving datagrams are just added to the socket receive buffer. But, this buffer has a limited size. We discussed this size and how to increase it with the SO_RCVBUF socket option in Section 7.5.

Figure 8.5 summarizes our TCP client/server from Chapter 5 when two clients establish connections with the server.

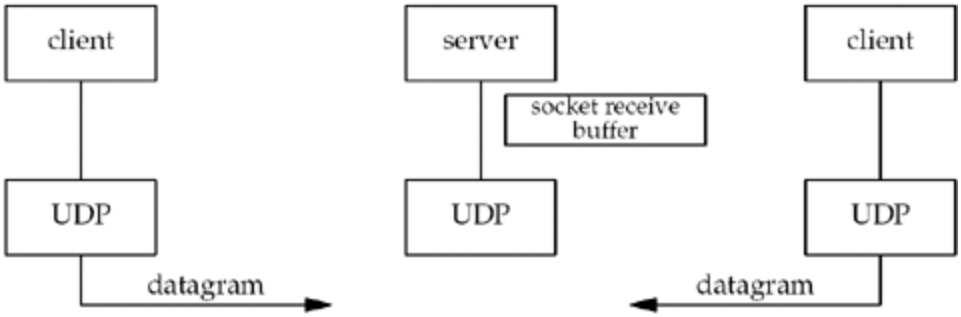
### Figure 8.5. Summary of TCP client/server with two clients.




There are two connected sockets and each of the two connected sockets on the server host has its own socket receive buffer.

Figure 8.6 shows the scenario when two clients send datagrams to our UDP server.

### Figure 8.6. Summary of UDP client/server with two clients.




There is only one server process and it has a single socket on which it receives all arriving datagrams and sends all responses. That socket has a receive buffer into which all arriving datagrams are placed.

The `main` function in Figure 8.3 is *protocol-dependent* (it creates a socket of protocol `AF_INET` and allocates and initializes an IPv4 socket address structure), but the `dg_echo` function is *protocol-independent*. The reason `dg_echo` is protocol-independent is because the caller (the `main` function in our case) must allocate a socket address structure of the correct size, and a pointer to this structure, along with its size, are passed as arguments to `dg_echo`. The function `dg_echo` never looks inside this protocol-dependent structure: It simply passes a pointer to the structure to `recvfrom` and `sendto`. `recvfrom` fills this structure with the IP address and port number of the client, and since the same pointer (`pcliaddr`) is then passed to `sendto` as the destination address, this is how the datagram is echoed back to the client that sent the datagram.



PREVIOUS NEXT

# 8.5 UDP Echo Client: `main` Function

The UDP client `main` function is shown in Figure 8.7.

### Figure 8.7 UDP echo client.

*udpcliserv/udpcli01.c*

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     struct sockaddr_in servaddr;

 7     if(argc != 2)
 8        err_quit("usage: udpcli <IPaddress>");

 9     bzero(&servaddr, sizeof(servaddr));
10     servaddr.sin_family = AF_INET;
11     servaddr.sin_port = htons(SERV_PORT);
12     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

13     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

14     dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

15     exit(0);
16 }
```

## Fill in socket address structure with server's address

*9–12* An IPv4 socket address structure is filled in with the IP address and port number of the server. This structure will be passed to `dg_cli`, specifying where to send datagrams.

*13–14* A UDP socket is created and the function `dg_cli` is called.

## 8.6 UDP Echo Client: dg_cli Function

Figure 8.8 shows the function dg_cli, which performs most of the client processing.

### Figure 8.8 dg_cli function: client processing loop.

*lib/dg_cli.c*

```
 1 #include     "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 8         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 9         n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

10         recvline[n] = 0;        /* null terminate */
11         Fputs(recvline, stdout);
12     }
13 }
```

*7–12* There are four steps in the client processing loop: read a line from standard input using fgets, send the line to the server using sendto, read back the server's echo using recvfrom, and print the echoed line to standard output using fputs.

Our client has not asked the kernel to assign an ephemeral port to its socket. (With a TCP client, we said the call to connect is where this takes place.) With a UDP socket, the first time the process calls sendto, if the socket has not yet had a local port bound to it, that is when an ephemeral port is chosen by the kernel for the socket. As with TCP, the client can call bind explicitly, but this is rarely done.

Notice that the call to recvfrom specifies a null pointer as the fifth and sixth arguments. This tells the kernel that we are not interested in knowing who sent the reply. There is a risk that any process, on either the same host or some other host, can send a datagram to the client's IP address and port, and that datagram will be read by the client, who will think it is the server's reply. We will address this in Section 8.8.

As with the server function dg_echo, the client function dg_cli is protocol-independent, but the client main function is protocol-dependent. The main function allocates and initializes a socket address structure of some protocol type and then passes a pointer to this structure, along with its size, to dg_cli.

## 8.7 Lost Datagrams

Our UDP client/server example is not reliable. If a client datagram is lost (say it is discarded by some router between the client and server), the client will block forever in its call to `recvfrom` in the function `dg_cli`, waiting for a server reply that will never arrive. Similarly, if the client datagram arrives at the server but the server's reply is lost, the client will again block forever in its call to `recvfrom`. A typical way to prevent this is to place a timeout on the client's call to `recvfrom`. We will discuss this in Section 14.2.

Just placing a timeout on the `recvfrom` is not the entire solution. For example, if we do time out, we cannot tell whether our datagram never made it to the server, or if the server's reply never made it back. If the client's request was something like "transfer a certain amount of money from account A to account B" (instead of our simple echo server), it would make a big difference as to whether the request was lost or the reply was lost. We will talk more about adding reliability to a UDP client/server in Section 22.5.

# 8.8 Verifying Received Response

At the end of Section 8.6, we mentioned that any process that knows the client's ephemeral port number could send datagrams to our client, and these would be intermixed with the normal server replies. What we can do is change the call to `recvfrom` in Figure 8.8 to return the IP address and port of who sent the reply and ignore any received datagrams that are not from the server to whom we sent the datagram. There are a few pitfalls with this, however, as we will see.

First, we change the client `main` function (Figure 8.7) to use the standard echo server (Figure 2.18). We just replace the assignment

```
servaddr.sin_port = htons(SERV_PORT);
```

with

```
servaddr.sin_port = htons(7);
```

We do this so we can use any host running the standard echo server with our client.

We then recode the `dg_cli` function to allocate another socket address structure to hold the structure returned by `recvfrom`. We show this in Figure 8.9.

## Allocate another socket address structure

*9* We allocate another socket address structure by calling `malloc`. Notice that the `dg_cli` function is still protocol-independent; because we do not care what type of socket address structure we are dealing with, we use only its size in the call to `malloc`.

## Compare returned address

*12–18* In the call to `recvfrom`, we tell the kernel to return the address of the sender of the datagram. We first compare the length returned by `recvfrom` in the value-result argument and then compare the socket address structures themselves using `memcmp`.

> Section 3.2 says that even if the socket address structure contains a length field, we need never set it or examine it. However, `memcmp` compares every byte of data in the two socket address structures, and the length field is set in the socket address structure that the kernel returns; so in this case we must set it when constructing the `sockaddr`. If we don't, the `memcmp` will compare a *0* (since we didn't set it) with a *16* (assuming `sockaddr_in`) and will not match.

### Figure 8.9 Version of `dg_cli` that verifies returned socket address.

*udpcliserv/dgcliaddr.c*

```
 1 #include     "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];
 7     socklen_t len;
 8     struct sockaddr *preply_addr;

 9     preply_addr = Malloc(servlen);

10     while (Fgets(sendline, MAXLINE, fp) != NULL) {

11         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

12         len = servlen;
13         n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
14         if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
15             printf("reply from %s (ignored)\n", Sock_ntop(preply_addr, len));
16             continue;
17         }

18         recvline[n] = 0;      /* null terminate */
19         Fputs(recvline, stdout);
20     }
```

```
21 }
```

This new version of our client works fine if the server is on a host with just a single IP address. But this program can fail if the server is multihomed. We run this program to our host `freebsd4`, which has two interfaces and two IP addresses.

```
macosx % host freebsd4
freebsd4.unpbook.com has address 172.24.37.94
freebsd4.unpbook.com has address 135.197.17.100
macosx % udpcli02 135.197.17.100
hello
reply from 172.24.37.94:7 (ignored)
goodbye
reply from 172.24.37.94:7 (ignored)
```

We specified the IP address that does not share the same subnet as the client.

> This is normally allowed. Most IP implementations accept an arriving IP datagram that is destined for *any* of the host's IP addresses, regardless of the interface on which the datagram arrives (pp. 217–219 of TCPv2). RFC 1122 [Braden 1989] calls this the *weak end system model*. If a system implemented what this RFC calls the *strong end system model*, it would accept an arriving datagram only if that datagram arrived on the interface to which it was addressed.

The IP address returned by `recvfrom` (the source IP address of the UDP datagram) is not the IP address to which we sent the datagram. When the server sends its reply, the destination IP address is 172.24.37.78. The routing function within the kernel on `freebsd4` chooses 172.24.37.94 as the outgoing interface. Since the server has not bound an IP address to its socket (the server has bound the wildcard address to its socket, which is something we can verify by running `netstat` on `freebsd`), the kernel chooses the source address for the IP datagram. It is chosen to be the primary IP address of the outgoing interface (pp. 232–233 of TCPv2). Also, since it is the primary IP address of the interface, if we send our datagram to a nonprimary IP address of the interface (i.e., an alias), this will also cause our test in Figure 8.9 to fail.

One solution is for the client to verify the responding host's domain name instead of its IP address by looking up the server's name in the DNS (Chapter 11), given the IP address returned by `recvfrom`. Another solution is for the UDP server to create one socket for every IP address that is configured on the host, `bind` that IP address to the socket, use `select` across all these sockets (waiting for any one to become readable), and then reply from the socket that is readable. Since the socket used for the reply was bound to the IP address that was the destination address of the client's request (or the datagram would not have been delivered to the socket), this guaranteed that the source address of the reply was the same as the destination address of the request. We will show an example of this in Section 22.6.

> On a multihomed Solaris system, the source IP address for the server's reply is the destination IP address of the client's request. The scenario described in this section is for Berkeley-derived implementations that choose the source IP address based on the outgoing interface.

## 8.9 Server Not Running

The next scenario to examine is starting the client without starting the server. If we do so and type in a single line to the client, nothing happens. The client blocks forever in its call to `recvfrom`, waiting for a server reply that will never appear. But, this is an example where we need to understand more about the underlying protocols to understand what is happening to our networking application.

First we start `tcpdump` on the host `macosx`, and then we start the client on the same host, specifying the host `freebsd4` as the server host. We then type a single line, but the line is not echoed.

```
macosx % udpcli01 172.24.37.94
hello, world
```

*we type this line but nothing is echoed back*

Figure 8.10 shows the `tcpdump` output.

### Figure 8.10 `tcpdump` output when server process not started on server host.

```
1 0.0                   arp who-has freebsd4 tell macosx
2 0.003576 ( 0.0036)    arp reply freebsd4 is-at 0:40:5:42:d6:de

3 0.003601 ( 0.0000)    macosx.51139 > freebsd4.9877: udp 13
4 0.009781 ( 0.0062)    freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

First we notice that an ARP request and reply are needed before the client host can send the UDP datagram to the server host. (We left this exchange in the output to reiterate the potential for an ARP request-reply before an IP datagram can be sent to another host or router on the local network.)

In line 3, we see the client datagram sent but the server host responds in line 4 with an ICMP "port unreachable." (The length of 13 accounts for the 12 characters and the newline.) This ICMP error, however, is not returned to the client process, for reasons that we will describe shortly. Instead, the client blocks forever in the call to `recvfrom` in Figure 8.8. We also note that ICMPv6 has a "port unreachable" error, similar to ICMPv4 (Figures A.15 and A.16), so the results described here are similar for IPv6.

We call this ICMP error an *asynchronous error*. The error was caused by `sendto`, but `sendto` returned successfully. Recall from Section 2.11 that a successful return from a UDP output operation only means there was room for the resulting IP datagram on the interface output queue. The ICMP error is not returned until later (4 ms later in Figure 8.10), which is why it is called asynchronous.

The basic rule is that an asynchronous error is not returned for a UDP socket unless the socket has been connected. We will describe how to call `connect` for a UDP socket in Section 8.11. Why this design decision was made when sockets were first implemented is rarely understood. (The implementation implications are discussed on pp. 748–749 of TCPv2.)

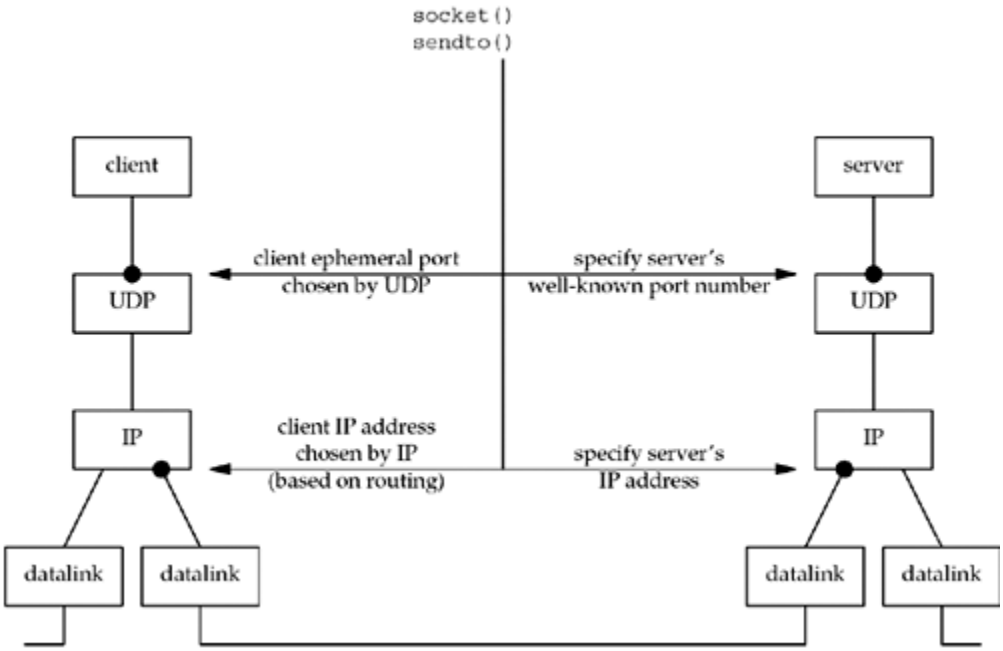
Consider a UDP client that sends three datagrams in a row to three different servers (i.e., three different IP addresses) on a single UDP socket. The client then enters a loop that calls `recvfrom` to read the replies. Two of the datagrams are correctly delivered (that is, the server was running on two of the three hosts) but the third host was not running the server. This third host responds with an ICMP port unreachable. This ICMP error message contains the IP header and UDP header of the datagram that caused the error. (ICMPv4 and ICMPv6 error messages always contain the IP header and all of the UDP header or part of the TCP header to allow the receiver of the ICMP error to determine which socket caused the error. We will show this in Figures 28.21 and 28.22.) The client that sent the three datagrams needs to know the destination of the datagram that caused the error to distinguish which of the three datagrams caused the error. But how can the kernel return this information to the process? The only piece of information that `recvfrom` can return is an `errno` value; `recvfrom` has no way of returning the destination IP address and destination UDP port number of the datagram in error. The decision was made, therefore, to return these asynchronous errors to the process only if the process connected the UDP socket to exactly one peer.

> Linux returns most ICMP "destination unreachable" errors even for unconnected sockets, as long as the `SO_BSDCOMPAT` socket option is not enabled. All the ICMP "destination unreachable" errors from Figure A.15 are returned, except codes 0, 1, 4, 5, 11, and 12.
>
> We return to this problem of asynchronous errors with UDP sockets in Section 28.7 and show an easy way to obtain these errors on unconnected sockets using a daemon of our own.

## 8.10 Summary of UDP Example

Figure 8.11 shows as bullets the four values that must be specified or chosen when the client sends a UDP datagram.

**Figure 8.11. Summary of UDP client/server from client's perspective.**

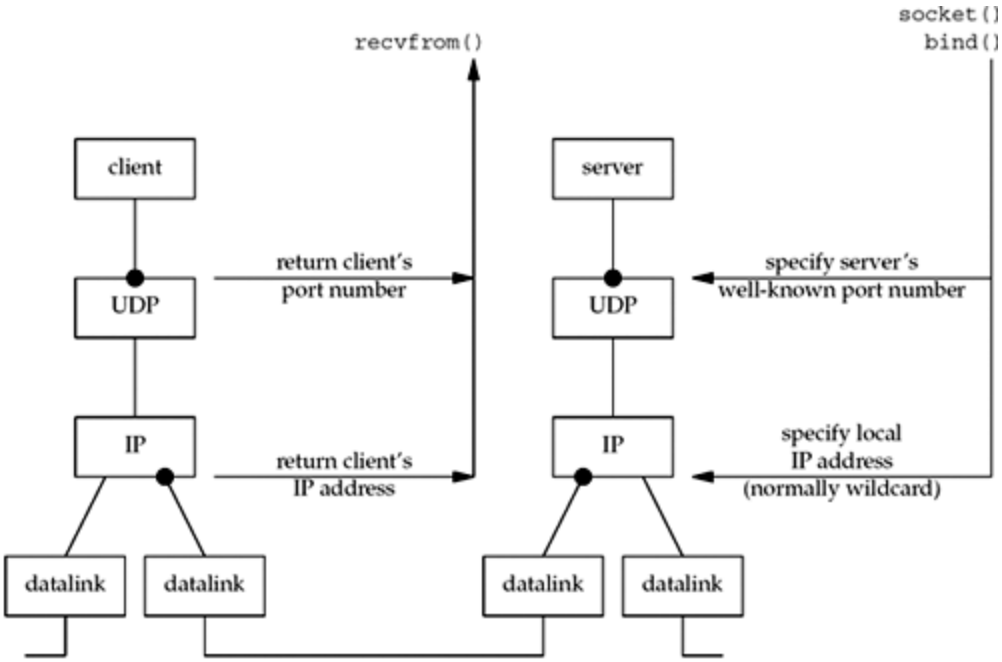



The client must specify the server's IP address and port number for the call to `sendto`. Normally, the client's IP address and port are chosen automatically by the kernel, although we mentioned that the client can call `bind` if it so chooses. If these two values for the client are chosen by the kernel, we also mentioned that the client's ephemeral port is chosen once, on the first `sendto`, and then it never changes. The client's IP address, however, can change for every UDP datagram that the client sends, assuming the client does not `bind` a specific IP address to the socket. The reason is shown in Figure 8.11: If the client host is multihomed, the client could alternate between two destinations, one going out the datalink on the left, and the other going out the datalink on the right. In this worst-case scenario, the client's IP address, as chosen by the kernel based on the outgoing datalink, would change for every datagram.

What happens if the client `binds` an IP address to its socket, but the kernel decides that an outgoing datagram must be sent out some other datalink? In this case the IP datagram will contain a source IP address that is different from the IP address of the outgoing datalink (see Exercise 8.6).

Figure 8.12 shows the same four values, but from the server's perspective.

**Figure 8.12. Summary of UDP client/server from server's perspective.**




There are at least four pieces of information that a server might want to know from an arriving IP datagram: the source IP address, destination IP address, source port number, and destination port number. Figure 8.13 shows the function calls that return this information for a TCP server and a UDP server.

**Figure 8.13. Information available to server from arriving IP datagram.**

| From client's IP datagram | TCP server | UDP server |
|---|---|---|
| Source IP address | `accept` | `recvfrom` |
| Source port number | `accept` | `recvfrom` |
| Destination IP address | `getsockname` | `recvmsg` |
| Destination port number | `getsockname` | `getsockname` |

A TCP server always has easy access to all four pieces of information for a connected socket, and these four values remain constant for the lifetime of a connection. With a UDP socket, however, the destination IP address can only be obtained by setting the `IP_RECVDSTADDR` socket option for IPv4 or the `IPV6_PKTINFO` socket option for IPv6 and then calling `recvmsg` instead of `recvfrom`. Since UDP is connectionless, the destination IP address can change for each datagram that is sent to the server. A UDP server can also receive datagrams destined for one of the host's broadcast addresses or for a multicast address, as we will discuss in Chapters 20 and 21. We will show how to determine the destination address of a UDP datagram in Section 22.2, after we cover the `recvmsg` function.

## 8.11 `connect` Function with UDP

We mentioned at the end of Section 8.9 that an asynchronous error is not returned on a UDP socket unless the socket has been connected. Indeed, we are able to call `connect` (Section 4.3) for a UDP socket. But this does not result in anything like a TCP connection: There is no three-way handshake. Instead, the kernel just checks for any immediate errors (e.g., an obviously unreachable destination), records the IP address and port number of the peer (from the socket address structure passed to `connect`), and returns immediately to the calling process.

> Overloading the `connect` function with this capability for UDP sockets is confusing. If the convention that `sockname` is the local protocol address and `peername` is the foreign protocol address is used, then a better name would have been `setpeername`. Similarly, a better name for the `bind` function would be `setsockname`.

With this capability, we must now distinguish between

- An *unconnected UDP socket*, the default when we create a UDP socket
- A *connected UDP socket*, the result of calling `connect` on a UDP socket

With a connected UDP socket, three things change, compared to the default unconnected UDP socket:

1. We can no longer specify the destination IP address and port for an output operation. That is, we do not use `sendto`, but `write` or `send` instead. Anything written to a connected UDP socket is automatically sent to the protocol address (e.g., IP address and port) specified by `connect`.

   > Similar to TCP, we can call `sendto` for a connected UDP socket, but we cannot specify a destination address. The fifth argument to `sendto` (the pointer to the socket address structure) must be a null pointer, and the sixth argument (the size of the socket address structure) should be 0. The POSIX specification states that when the fifth argument is a null pointer, the sixth argument is ignored.

2. We do not need to use `recvfrom` to learn the sender of a datagram, but `read`, `recv`, or `recvmsg` instead. The only datagrams returned by the kernel for an input operation on a connected UDP socket are those arriving from the protocol address specified in `connect`. Datagrams destined to the connected UDP socket's local protocol address (e.g., IP address and port) but arriving from a protocol address other than the one to which the socket was connected are not passed to the connected socket. This limits a connected UDP socket to exchanging datagrams with one and only one peer.

   > Technically, a connected UDP socket exchanges datagrams with only one IP address, because it is possible to `connect` to a multicast or broadcast address.

3. Asynchronous errors are returned to the process for connected UDP sockets.

   The corollary, as we previously described, is that unconnected UDP sockets do not receive asynchronous errors.

Figure 8.14 summarizes the first point in the list with respect to 4.4BSD.

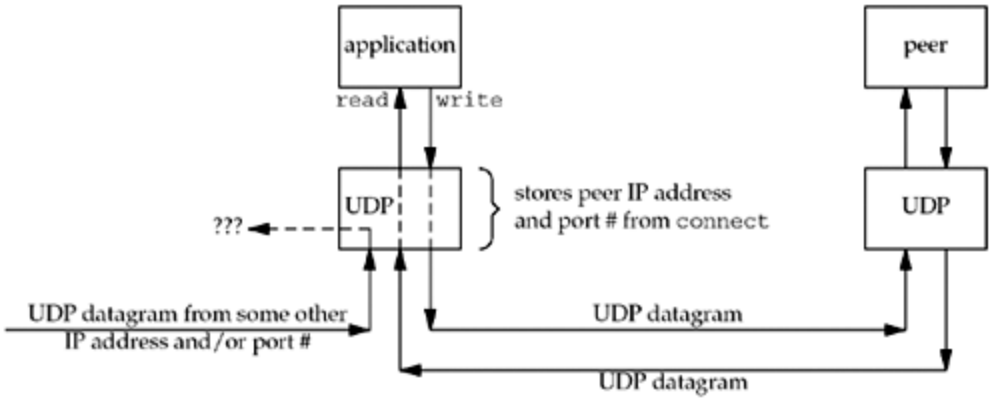
**Figure 8.14. TCP and UDP sockets: can a destination protocol address be specified?**

| Type of socket | `write` or `send` | `sendto` that does not specify a destination | `sendto` that specifies a destination |
|---|---|---|---|
| TCP socket | OK | OK | `EISCONN` |
| UDP socket, connected | OK | OK | `EISCONN` |
| UDP socket, unconnected | `EDESTADDRREQ` | `EDESTADDRREQ` | OK |

The POSIX specification states that an output operation that does not specify a destination address on an unconnected UDP socket should return `ENOTCONN`, not `EDESTADDRREQ`.

Figure 8.15 summarizes the three points that we made about a connected UDP socket.

**Figure 8.15. Connected UDP socket.**



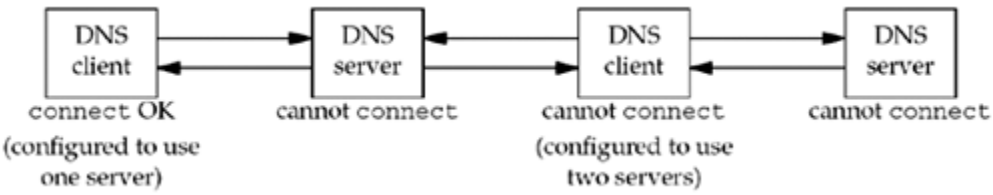

The application calls `connect`, specifying the IP address and port number of its peer. It then uses `read` and `write` to exchange data with the peer.

Datagrams arriving from any other IP address or port (which we show as "???" in Figure 8.15) are not passed to the connected socket because either the source IP address or source UDP port does not match the protocol address to which the socket is `connected`. These datagrams could be delivered to some other UDP socket on the host. If there is no other matching socket for the arriving datagram, UDP will discard it and generate an ICMP "port unreachable" error.

In summary, we can say that a UDP client or server can call `connect` only if that process uses the UDP socket to communicate with exactly one peer. Normally, it is a UDP client that calls `connect`, but there are applications in which the UDP server communicates with a single client for a long duration (e.g., TFTP); in this case, both the client and server can call `connect`.

The DNS provides another example, as shown in Figure 8.16.

**Figure 8.16. Example of DNS clients and servers and the `connect` function.**




A DNS client can be configured to use one or more servers, normally by listing the IP addresses of the servers in the file `/etc/resolv.conf`. If a single server is listed (the leftmost box in the figure), the client can call `connect`, but if multiple servers are listed (the second box from the right in the figure), the client cannot call `connect`. Also, a DNS server normally handles any client request, so the servers cannot call `connect`.

## Calling `connect` Multiple Times for a UDP Socket

A process with a connected UDP socket can call `connect` again for that socket for one of two reasons:

- To specify a new IP address and port
- To unconnect the socket

The first case, specifying a new peer for a connected UDP socket, differs from the use of `connect` with a TCP socket: `connect` can be called only one time for a TCP socket.

To unconnect a UDP socket, we call `connect` but set the family member of the socket address structure (`sin_family` for IPv4 or `sin6_family` for IPv6) to `AF_UNSPEC`. This might return an error of `EAFNOSUPPORT` (p. 736 of TCPv2), but that is acceptable. It is the process of calling `connect` on an already connected UDP socket that causes the socket to become unconnected (pp. 787–788 of TCPv2).

> The Unix variants seem to differ on exactly how to unconnect a socket, and you may encounter approaches that work on some systems and not others. For example, calling `connect` with `NULL` for the address works only on some systems (and on some, it only works if the third argument, the length, is nonzero). The POSIX specification and BSD man pages are not much help here, only mentioning that a *null address* should be used and not mentioning the error return (even on success) at all. The most portable solution is to zero out an address structure, set the family to `AF_UNSPEC` as mentioned above, and pass it to `connect`.
>
> Another area of disagreement is around the local binding of a socket during the unconnect process. AIX keeps both the chosen local IP address and the port, even from an implicit bind. FreeBSD and Linux set the local IP address back to all zeros, even if you previously called `bind`, but leave the port number intact. Solaris sets the local IP address back to all zeros if it was

assigned by an implicit bind; but if the program called `bind` explicitly, then the IP address remains unchanged.

## Performance

When an application calls `sendto` on an unconnected UDP socket, Berkeley-derived kernels temporarily connect the socket, send the datagram, and then unconnect the socket (pp. 762–763 of TCPv2). Calling `sendto` for two datagrams on an unconnected UDP socket then involves the following six steps by the kernel:

- Connect the socket
- Output the first datagram
- Unconnect the socket
- Connect the socket
- Output the second datagram
- Unconnect the socket

Another consideration is the number of searches of the routing table. The first temporary connect searches the routing table for the destination IP address and saves (caches) that information. The second temporary connect notices that the destination address equals the destination of the cached routing table information (we are assuming two `sendtos` to the same destination) and we do not need to search the routing table again (pp. 737–738 of TCPv2).

When the application knows it will be sending multiple datagrams to the same peer, it is more efficient to connect the socket explicitly. Calling `connect` and then calling `write` two times involves the following steps by the kernel:

- Connect the socket
- Output first datagram
- Output second datagram

In this case, the kernel copies only the socket address structure containing the destination IP address and port one time, versus two times when `sendto` is called twice. [Partridge and Pink 1993] note that the temporary connecting of an unconnected UDP socket accounts for nearly one-third of the cost of each UDP transmission.



PREVIOUS NEXT

## 8.12 dg_cli Function (Revisited)

We now return to the dg_cli function from Figure 8.8 and recode it to call connect. Figure 8.17 shows the new function.

### Figure 8.17 dg_cli function that calls connect.

*udpcliserv/dgcliconnect.c*

```
 1 #include     "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5     int     n;
 6     char    sendline[MAXLINE], recvline[MAXLINE + 1];

 7     Connect(sockfd, (SA *) pservaddr, servlen);

 8     while (Fgets(sendline, MAXLINE, fp) != NULL) {

 9         Write(sockfd, sendline, strlen(sendline));

10         n = Read(sockfd, recvline, MAXLINE);

11         recvline[n] = 0;        /* null terminate */
12         Fputs(recvline, stdout);
13     }
14 }
```

The changes are the new call to connect and replacing the calls to sendto and recvfrom with calls to write and read. This function is still protocol-independent since it doesn't look inside the socket address structure that is passed to connect. Our client main function, Figure 8.7, remains the same.

If we run this program on the host macosx, specifying the IP address of the host freebsd4 (which is not running our server on port 9877), we have the following output:

```
macosx % udpcli04 172.24.37.94
hello, world
read error: Connection refused
```

The first point we notice is that we do *not* receive the error when we start the client process. The error occurs only after we send the first datagram to the server. It is sending this datagram that elicits the ICMP error from the server host. But when a TCP client calls connect, specifying a server host that is not running the server process, connect returns the error because the call to connect causes the TCP three-way handshake to happen, and the first packet of that handshake elicits an RST from the server TCP (Section 4.3).

Figure 8.18 shows the tcpdump output.

### Figure 8.18 tcpdump output when running Figure 8.17.

```
macosx % tcpdump
1  0.0                    macosx.51139 > freebsd4.9877: udp 13
2  0.006180 ( 0.0062)     freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

We also see in Figure A.15 that this ICMP error is mapped by the kernel into the error ECONNREFUSED, which corresponds to the message string output by our err_sys function: "Connection refused."

> Unfortunately, not all kernels return ICMP messages to a connected UDP socket, as we have shown in this section. Normally, Berkeley-derived kernels return the error, while System V kernels do not. For example, if we run the same client on a Solaris 2.4 host and connect to a host that is not running our server, we can watch with tcpdump and verify that the ICMP "port unreachable" error is returned by the server host, but the client's call to read never returns. This bug was fixed in Solaris 2.5. UnixWare does not return the error, while AIX, Digital Unix, HP-UX, and Linux all return the error.

## 8.13 Lack of Flow Control with UDP

We now examine the effect of UDP not having any flow control. First, we modify our `dg_cli` function to send a fixed number of datagrams. It no longer reads from standard input. Figure 8.19 shows the new version. This function writes 2,000 1,400-byte UDP datagrams to the server.

We next modify the server to receive datagrams and count the number received. This server no longer echoes datagrams back to the client. Figure 8.20 shows the new `dg_echo` function. When we terminate the server with our terminal interrupt key (`SIGINT`), it prints the number of received datagrams and terminates.

### Figure 8.19 `dg_cli` function that writes a fixed number of datagrams to the server.

*udpcliserv/dgcliloop1.c*

```
 1 #include     "unp.h"

 2 #define NDG      2000          /*  datagrams to send */
 3 #define DGLEN    1400          /*  length of each datagram */

 4 void
 5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 6 {
 7     int     i;
 8     char    sendline[DGLEN];

 9     for (i = 0; i < NDG; i++) {
10         Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
11     }
12 }
```

### Figure 8.20 `dg_echo` function that counts received datagrams.

*udpcliserv/dgecholoop1.c*

```
 1 #include     "unp.h"

 2 static void recvfrom_int(int);
 3 static int count;

 4 void
 5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 6 {
 7     socklen_t len;
 8     char    mesg[MAXLINE];

 9     Signal(SIGINT, recvfrom_int);

10     for ( ; ; ) {
11         len = clilen;
12         Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

13         count++;
14     }
15 }

16 static void
17 recvfrom_int(int signo)
18 {
19     printf("\nreceived %d datagrams\n", count);
20     exit(0);
21 }
```

We now run the server on the host `freebsd`, a slow SPARCStation. We run the client on the RS/6000 system `aix`, connected directly with 100Mbps Ethernet. Additionally, we run `netstat -s` on the server, both before and after, as the statistics that are output tell us how many datagrams were lost. Figure 8.21 shows the output on the server.

### Figure 8.21 Output on server host.

```
freebsd % netstat -s -p udp
udp:
```

```
        71208 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        0 with no checksum
        832 dropped due to no socket
        16 broadcast/multicast datagrams dropped due to no socket
        1971 dropped due to full socket buffers
        0 not for hashed pcb
        68389 delivered
        137685 datagrams output
freebsd % udpserv06             start our server
                                                                    we run the client here
     ^C                                 we type our interrupt key after the client is finished
received 30 datagrams
freebsd % netstat -s -p udp
udp:
        73208 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        0 with no checksum
        832 dropped due to no socket
        16 broadcast/multicast datagrams dropped due to no socket
        3941 dropped due to full socket buffers
        0 not for hashed pcb
        68419 delivered
        137685 datagrams output
```

The client sent 2,000 datagrams, but the server application received only 30 of these, for a 98% loss rate. There is *no* indication whatsoever to the server application or to the client application that these datagrams were lost. As we have said, UDP has no flow control and it is unreliable. It is trivial, as we have shown, for a UDP sender to overrun the receiver.

If we look at the `netstat` output, the total number of datagrams received by the server host (not the server application) is 2,000 (73,208 - 71,208). The counter "dropped due to full socket buffers" indicates how many datagrams were received by UDP but were discarded because the receiving socket's receive queue was full (p. 775 of TCPv2). This value is 1,970 (3,491 - 1,971), which when added to the counter output by the application (30), equals the 2,000 datagrams received by the host. Unfortunately, the `netstat` counter of the number dropped due to a full socket buffer is systemwide. There is no way to determine which applications (e.g., which UDP ports) are affected.

The number of datagrams received by the server in this example is not predictable. It depends on many factors, such as the network load, the processing load on the client host, and the processing load on the server host.

If we run the same client and server, but this time with the client on the slow Sun and the server on the faster RS/6000, no datagrams are lost.

```
aix % udpserv06

^?                                                  we type our interrupt key after the client is finished

received 2000 datagrams
```

## UDP Socket Receive Buffer

The number of UDP datagrams that are queued by UDP for a given socket is limited by the size of that socket's receive buffer. We can change this with the `SO_RCVBUF` socket option, as we described in Section 7.5. The default size of the UDP socket receive buffer under FreeBSD is 42,080 bytes, which allows room for only 30 of our 1,400-byte datagrams. If we increase the size of the socket receive buffer, we expect the server to receive additional datagrams. Figure 8.22 shows a modification to the `dg_echo` function from Figure 8.20 that sets the socket receive buffer to 240 KB.

### Figure 8.22 `dg_echo` function that increases the size of the socket receive queue.

*udpcliserv/dgecholoop2.c*

```
 1 #include    "unp.h"

 2 static void recvfrom_int(int);
 3 static int count;

 4 void
 5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 6 {
 7     int     n;
 8     socklen_t len;
 9     char    mesg[MAXLINE];

10     Signal(SIGINT, recvfrom_int);
```

```
11      n = 220 * 1024;
12      Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

13      for ( ; ; ) {
14          len = clilen;
15          Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

16          count++;
17      }
18  }

19  static void
20  recvfrom_int(int signo)
21  {
22      printf("\nreceived %d datagrams\n", count);
23      exit(0);
24  }
```

If we run this server on the Sun and the client on the RS/6000, the count of received datagrams is now 103. While this is slightly better than the earlier example with the default socket receive buffer, it is no panacea.

> Why do we set the receive socket buffer size to 220 x 1,024 in Figure 8.22? The maximum size of a socket receive buffer in FreeBSD 5.1 defaults to 262,144 bytes (256 x 1,024), but due to the buffer allocation policy (described in Chapter 2 of TCPv2), the actual limit is 233,016 bytes. Many earlier systems based on 4.3BSD restricted the size of a socket buffer to around 52,000 bytes.

[ Team LiB ]

PREVIOUS NEXT

# 8.14 Determining Outgoing Interface with UDP

A connected UDP socket can also be used to determine the outgoing interface that will be used to a particular destination. This is because of a side effect of the `connect` function when applied to a UDP socket: The kernel chooses the local IP address (assuming the process has not already called `bind` to explicitly assign this). This local IP address is chosen by searching the routing table for the destination IP address, and then using the primary IP address for the resulting interface.

Figure 8.23 shows a simple UDP program that `connects` to a specified IP address and then calls `getsockname`, printing the local IP address and port.

### Figure 8.23 UDP program that uses `connect` to determine outgoing interface.

*udpcliserv/udpcli09.c*

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     socklen_t len;
 7     struct sockaddr_in cliaddr, servaddr;

 8     if (argc != 2)
 9         err_quit("usage: udpcli <IPaddress>");

10     sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(SERV_PORT);
14     Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

15     Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

16     len = sizeof(cliaddr);
17     Getsockname(sockfd, (SA *) &cliaddr, &len);
18     printf("local address %s\n", Sock_ntop((SA *) &cliaddr, len));

19     exit(0);
20 }
```

If we run the program on the multihomed host `freebsd`, we have the following output:

```
freebsd % udpcli09 206.168.112.96
local address 12.106.32.254:52329

freebsd % udpcli09 192.168.42.2
local address 192.168.42.1:52330

freebsd % udpcli09 127.0.0.1
local address 127.0.0.1:52331
```

The first time we run the program, the command-line argument is an IP address that follows the default route. The kernel assigns the local IP address to the primary address of the interface to which the default route points. The second time, the argument is the IP address of a system connected to a second Ethernet interface, so the kernel assigns the local IP address to the primary address of this second interface. Calling `connect` on a UDP socket does not send anything to that host; it is entirely a local operation that saves the peer's IP address and port. We also see that calling `connect` on an unbound UDP socket also assigns an ephemeral port to the socket.

> Unfortunately, this technique does not work on all implementations, mostly SVR4-derived kernels. For example, this does not work on Solaris 2.5, but it works on AIX, HP-UX 11, MacOS X, FreeBSD, Linux, and Solaris 2.6 and later.

# 8.15 TCP and UDP Echo Server Using `select`

We now combine our concurrent TCP echo server from Chapter 5 with our iterative UDP echo server from this chapter into a single server that uses `select` to multiplex a TCP and UDP socket. Figure 8.24 is the first half of this server.

## Create listening TCP socket

*14–22* A listening TCP socket is created that is bound to the server's well-known port. We set the `SO_REUSEADDR` socket option in case connections exist on this port.

## Create UDP socket

*23–29* A UDP socket is also created and bound to the same port. Even though the same port is used for TCP and UDP sockets, there is no need to set the `SO_REUSEADDR` socket option before this call to `bind`, because TCP ports are independent of UDP ports.

Figure 8.25 shows the second half of our server.

### Establish signal handler for `SIGCHLD`

*30* A signal handler is established for `SIGCHLD` because TCP connections will be handled by a child process. We showed this signal handler in Figure 5.11.

### Prepare for `select`

*31–32* We initialize a descriptor set for `select` and calculate the maximum of the two descriptors for which we will wait.

### Figure 8.24 First half of echo server that handles TCP and UDP using `select`.

*udpcliserv/udpservselect01.c*

```
 1 #include     "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     listenfd, connfd, udpfd, nready, maxfdp1;
 6     char    mesg[MAXLINE];
 7     pid_t   childpid;
 8     fd_set  rset;
 9     ssize_t n;
10     socklen_t len;
11     const int on = 1;
12     struct sockaddr_in cliaddr, servaddr;
13     void    sig_chld(int);

14         /* create listening TCP socket */
15     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

16     bzero(&servaddr, sizeof(servaddr));
17     servaddr.sin_family = AF_INET;
18     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19     servaddr.sin_port = htons(SERV_PORT);

20     Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

22     Listen(listenfd, LISTENQ);

23          /* create UDP socket */
24     udpfd = Socket(AF_INET, SOCK_DGRAM, 0);

25     bzero(&servaddr, sizeof(servaddr));
26     servaddr.sin_family = AF_INET;
27     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
28      servaddr.sin_port = htons(SERV_PORT);

29      Bind(udpfd, (SA *) &servaddr, sizeof(servaddr));
```

### Call `select`

*34–41* We call `select`, waiting only for readability on the listening TCP socket or readability on the UDP socket. Since our `sig_chld` handler can interrupt our call to `select`, we handle an error of `EINTR`.

### Handle new client connection

*42–51* We `accept` a new client connection when the listening TCP socket is readable, `fork` a child, and call our `str_echo` function in the child. This is the same sequence of steps we used in Chapter 5.

### Figure 8.25 Second half of echo server that handles TCP and UDP using `select`.

*udpcliserv/udpservselect01.c*

```
30      Signal(SIGCHLD, sig_chld);      /* must call waitpid() */

31      FD_ZERO(&rset);
32      maxfdp1 = max(listenfd, udpfd) + 1;
33      for ( ; ; ) {
34          FD_SET(listenfd, &rset);
35          FD_SET(udpfd, &rset);
36          if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
37              if (errno == EINTR)
38                  continue;      /* back to for() */
39              else
40                  err_sys("select error");
41           }

42           if (FD_ISSET(listenfd, &rset)) {
43               len = sizeof(cliaddr);
44               connfd = Accept(listenfd, (SA *) &cliaddr, &len);

45               if ( (childpid = Fork()) == 0) { /* child process */
46                   Close(listenfd);      /* close listening socket */
47                   str_echo(connfd);     /* process the request */
48                   exit(0);
49                }
50                Close(connfd);      /* parent closes connected socket */
51          }

52          if (FD_ISSET(udpfd, &rset)) {
53              len = sizeof(cliaddr);
54              n = Recvfrom(udpfd, mesg, MAXLINE, 0, (SA *) &cliaddr, &len);

55              Sendto(udpfd, mesg, n, 0, (SA *) &cliaddr, len);
56          }
57      }
58  }
```

### Handle arrival of datagram

*52–57* If the UDP socket is readable, a datagram has arrived. We read it with `recvfrom` and send it back to the client with `sendto`.

## 8.16 Summary

Converting our echo client/server to use UDP instead of TCP was simple. But lots of features provided by TCP are missing: detecting lost packets and retransmitting, verifying responses as being from the correct peer, and the like. We will return to this topic in Section 22.5 and see what it takes to add some reliability to a UDP application.

UDP sockets can generate asynchronous errors, that is, errors that are reported some time after a packet is sent. TCP sockets always report these errors to the application, but with UDP, the socket must be connected to receive these errors.

UDP has no flow control, and this is easy to demonstrate. Normally, this is not a problem, because many UDP applications are built using a request-reply model, and not for transferring bulk data.

There are still more points to consider when writing UDP applications, but we will save these until Chapter 22, after covering the interface functions, broadcasting, and multicasting.