## 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.