20.4 dg_cli Function Using Broadcasting

We modify our dg_cli function one more time, this time allowing it to broadcast to the standard UDP daytime server (Figure 2.18) and printing all replies. The only change we make to the main function (Figure 8.7) is to change the destination port number to 13.

```
servaddr.sin_port = htons(13);
```

We first compile this modified main function with the unmodified dg_cli function from Figure 8.8 and run it on the host freebsd.

```
freebsd % udpcli01 192.168.42.255
hi
sendto error: Permission denied
```

The command-line argument is the subnet-directed broadcast address for the secondary Ethernet. We type a line of input, the program calls sendto, and the error EACCES is returned. The reason we receive the error is that we are not allowed to send a datagram to a broadcast destination address unless we explicitly tell the kernel that we will be broadcasting. We do this by setting the SO_BROADCAST socket option (Section 7.5).

> Berkeley-derived implementations implement this sanity check. Solaris 2.5, on the other hand, accepts the datagram destined for the broadcast address even if we do not specify the socket option. The POSIX specification requires the SO_BROADCAST socket option to be set to send a broadcast packet.
>
> Broadcasting was a privileged operation with 4.2BSD and the SO_BROADCAST socket option did not exist. This option was added to 4.3BSD and any process was allowed to set the option.

We now modify our dg_cli function as shown in Figure 20.5. This version sets the SO_BROADCAST socket option and prints all the replies received within five seconds.

Allocate room for server's address, set socket option

11–13 malloc allocates room for the server's address to be returned by recvfrom. The SO_BROADCAST socket option is set and a signal handler is installed for SIGALRM.

Read line, send to socket, read all replies

14–24 The next two steps, fgets and sendto, are similar to previous versions of this function. But since we are sending a broadcast datagram, we can receive multiple replies. We call recvfrom in a loop and print all the replies received within five seconds. After five seconds, SIGALRM is generated, our signal handler is called, and recvfrom returns the error EINTR.

Print each received reply

25–29 For each reply received, we call sock_ntop_host, which in the case of IPv4 returns a string containing the dotted-decimal IP address of the server. This is printed along with the server's reply.

If we run the program specifying the subnet-directed broadcast address of 192.168.42.255, we see the following:

```
freebsd % udpcli01 192.168.42.255
hi
from 192.168.42.2: Sat Aug 2 16:42:45 2003
from 192.168.42.1: Sat Aug 2 14:42:45 2003
from 192.168.42.3: Sat Aug 2 14:42:45 2003
hello
from 192.168.42.3: Sat Aug 2 14:42:57 2003
from 192.168.42.2: Sat Aug 2 16:42:57 2003
from 192.168.42.1: Sat Aug 2 14:42:57 2003
```

Each time we must type a line of input to generate the output UDP datagram. Each time we receive three replies, and this includes the sending host. As we said earlier, the destination of a broadcast datagram is *all* the hosts on the attached network, including the sender. Each reply is unicast because the source address of the request, which is used by each server as the destination address of the reply, is a unicast address.

All the systems report the same time because all run NTP.

Figure 20.5 `dg_cli` function that broadcasts.

bcast/dgclibcast1.c

```
 1 #include      "unp.h"

 2 static void recvfrom_alarm(int);

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

11     preply_addr = Malloc(servlen);

12     Setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

13     Signal(SIGALRM, recvfrom_alarm);

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

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

16         alarm(5);
17         for ( ; ; ) {
18             len = servlen;
19             n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
20             if (n < 0) {
21                 if (errno == EINTR)
22                     break;      /* waited long enough for replies */
23                 else
24                    err_sys("recvfrom error");
25             } else {
26                 recvline[n] = 0; /* null terminate */
27                 printf("from %s: %s",
28                        Sock_ntop_host(preply_addr, len), recvline);
29             }
30         }
31     }
32     free(preply_addr);
33 }

34 static void
35 recvfrom_alarm(int signo)
36 {
37     return;                     /* just interrupt the recvfrom() */
38 }
```

IP Fragmentation and Broadcasts

Berkeley-derived kernels do not allow a broadcast datagram to be fragmented. If the size of an IP datagram that is being sent to a broadcast address exceeds the outgoing interface MTU, `EMSGSIZE` is returned (pp. 233–234 of TCPv2). This is a policy decision that has existed since 4.2BSD. There is nothing that prevents a kernel from fragmenting a broadcast datagram, but the feeling is that broadcasting puts enough load on the network as it is, so there is no need to multiply this load by the number of fragments.

We can see this scenario with our program in Figure 20.5. We redirect standard input from a file containing a 2,000-byte line, which will require fragmentation on an Ethernet.

```
freebsd % udpcli01 192.168.42.255 < 2000line
sendto error: Message too long
```

AIX, FreeBSD, and MacOS implement this limitation. Linux, Solaris, and HP-UX fragment datagrams sent to a broadcast address. For portability, however, an application that needs to broadcast should determine the MTU of the outgoing interface using the `SIOCGIFMTU ioctl`, and then subtract the IP and transport header lengths to determine the maximum payload size. Alternately, it can pick a common MTU, like Ethernet's 1500, and use it as a constant.