CHAPTER 8  ELEMENTARY UDP SOCKETS

User Datagram Protocol is a connectionless, “unreliable” datagram protocol.

Function calls for a typical UDP Client/server

- Client does not establish a connection with the server.
- Server does not accept a connection from a client.
- recvfrom function returns the protocol address of the client so server can send a response to correct client.
Topics

• Fundamental differences in the two transport layer protocols: TCP and UDP
• New functions that we use with UDP sockets (e.g., `recvfrom()` and `sendto()`)
• The use of `connect()` with UDP
• Concept of asynchronous errors
Introduction

- **User Datagram Protocol** is a connectionless, “unreliable” datagram protocol
- There are still instances when UDP is the protocol of choice (e.g., DNS, SNMP, etc.)
- Typically UDP clients don’t establish a connection with the server
- UDP server does not accept a connection from a client
Figure 8.1 Socket functions for UDP client–server.
8.2 recvfrom and sendto Functions

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

```c
#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

[ from p. 240 ]
recvfrom and sendto functions

 recvfrom

 sockfd -    Descriptor
 *buff    - Pointer to buffer to read or write.
nbytes   - Number of Bytes to read or write.
 flags   - Set to 0 for now

 sendto

to

 socket address structure containing the protocol
 address (IP address and port #) of where data is to be sent.

 addrlen  Size of socket address structure (an integer value).

 recvfrom

 from

 socket address structure that is filled in by recvfrom with the protocol address (IP
 address and port #) of where data came from.

 *addrlen ← ( pointer to integer value )
UDP Echo Server

Redo echo client - server from chapter 5 but using UDP
# UDP Echo Server: main Function

```
#include "unp.h"

int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_in servaddr, cliaddr;

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(SERV_PORT);

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

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

*Figure 8.3* UDP echo server.
8.4 UDP Echo Server: \texttt{dg\_echo} Function

Figure 8.4 shows the \texttt{dg\_echo} function.

\begin{verbatim}
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    }
\end{verbatim}

\textbf{Figure 8.4} \texttt{dg\_echo} function: echo lines on a datagram socket.
Server main Function

Lines 7-12:

Note now we use SOCK_DGRAM to mean UDP
INADDR_ANY = 0
SERV_PORT = 9877

Line 13:

dg_echo does the server processing

Server dg_echo function

Line 8-12:

Loop reads next datagram and sends back.
Note:

• This dg-echo function never terminates

• This is an iterative server which is typical for UDP. A concurrent server calls fork function, TCP is typically concurrent

• Each UDP socket has a receive buffer. (Aside: size is controlled by a socket option SO_RCVBUF)

*Difference between UDP and TCP implementations of our example
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.
UDP Echo Server: dg_echo function (Continued)

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.
UDP echo Client: main Function

```
#include "unp.h"

int main(int argc, char **argv)
{
    int sockfd;
    struct sockaddr_in servaddr;

    if (argc != 2)
        err_quit("usage: udpcli <IPaddress>"AtIndex10);

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

    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

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

exit(0);
```

Figure 8.7 UDP echo client.
UDP echo Client: dg_cli Function (Continued)

```c
#include "unp.h"

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

  while (Fgets(sendline, MAXLINE, fp) != NULL) {
    Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);
    n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
    recvline[n] = 0; /* null terminate */
    Fputs(recvline, stdout);
  }
```

Figure 8.8 dg_cli function: client processing loop.
UDP echo Client: main Function (Continued)

Lines 9-12: IPV4 socket address structure is filled in with the IP address, and port number of server

Lines 13: UDP socket is created

Lines 14: Call dg_cli

dg_cli function

Lines 7: Read a line from standard input using fgets
Lines 8: Send the line to server using sendto
Lines 9: Read back server’s ECHO using recvfrom
Lines 10: Print the echoed line to standard output using fputs

Note: Our client does not call bind, it is NOT NECESSARY
Summary of UDP Examples

Figure 8.11 Summary of UDP client/server from client’s perspective.
Summary of UDP Examples (Continued)

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

<table>
<thead>
<tr>
<th>From client’s IP datagram</th>
<th>TCP server</th>
<th>UDP server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source IP address</td>
<td>accept</td>
<td>recvfrom</td>
</tr>
<tr>
<td>Source port number</td>
<td>accept</td>
<td>recvfrom</td>
</tr>
<tr>
<td>Destination IP address</td>
<td>getsockname</td>
<td>recvmsg</td>
</tr>
<tr>
<td>Destination port number</td>
<td>getsockname</td>
<td>getsockname</td>
</tr>
</tbody>
</table>

Figure 8.13  Information available to server from arriving IP datagram.
connect Function with UDP

• Although UDP is a connectionless protocol, you can call `connect()` when using UDP

• This does not result in anything similar to TCP’s behavior when `connect` is called

• For UDP, the Kernel checks for any immediate errors, records the IP address and port number of the peer, and returns immediately to the calling process
connect Function with UDP (Continued)

• Unconnected UDP socket:
  – Default when we create a UDP socket
• Connected UDP socket:
  – Result of calling `connect()`
  – You no longer specify the destination IP and port for an output operation. That is, you must use `write` or `send` instead of `sendto`
  – Don’t need `recvfrom()`, rather `read()`, `recv()`, or `recvmsg()`
  – Asynchronous errors are returned to the process
  – Datagrams arriving from any other IP address and port are discarded
  – Client or server should call connect only when communicating with one peer
connect Function with UDP (Continued)

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

<table>
<thead>
<tr>
<th>Type of socket</th>
<th>write or send</th>
<th>sendto that does not specify a destination</th>
<th>sendto that specifies a destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP socket</td>
<td>OK</td>
<td>OK</td>
<td>EISCONN</td>
</tr>
<tr>
<td>UDP socket, connected</td>
<td>OK</td>
<td>EDESTADDRREQ</td>
<td>EISCONN</td>
</tr>
<tr>
<td>UDP socket, unconnected</td>
<td>EDESTADDRREQ</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

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

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.
connect Function with UDP (Continued)

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

- DNS client can be configured to use more than one server
- If a single server is listed, the client can call connect
- If multiple servers are listed, the client cannot call connect()
- Further, DNS servers normally handle any client request, to the servers cannot call connect()
- A process with a connected UDP socket can call connect() again for two reasons:
  - To specify a new IP address and port
  - To unconnect the socket
- When sending several datagrams to the same receiver, it's more efficient to use a connected socket
dg_cli Function (Revisited)

```c
#include "unp.h"

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

    while (fgets(sendline, MAXLINE, fp) != NULL) {
        Write(sockfd, sendline, strlen(sendline));
        n = Read(sockfd, recvline, MAXLINE);
        recvline[n] = 0;  /* null terminate */
        Fputs(recvline, stdout);
    }
}
```

Figure 8.17  dg_cli function that calls connect.
dg_cli Function (Revisited) - (Continued)

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

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

Figure 8.18 tcpdump output when running Figure 8.17.

• Changes:
  – call to `connect()`
  – replaced calls to `sendto()` and `recvfrom()` with calls to `write()` and `read()`

• Notice that we received an error when running the program and the first datagram is sent – How does this differ from TCP and an unconnected UDP socket?
Lack of Flow Control With UDP

- Modify client dg_cli function to send a fixed number of datagrams

- Now writes 2,000 times a 1,400 BYTE UDP datagram to server

- Modify server dg_echo function to receive datagrams and count the number received
  
  ( No longer echo datagrams back to client )

Line 9: Adds ability to execute the function recvfrom_int when type cntrl C
This function prints out how many we received
Lack of Flow Control With UDP (Continued)

```c
#include "unp.h"

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

void
dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
    int i;
    char sendline[MAXLINE];

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

Figure 8.19  dg_cli function that writes a fixed number of datagrams to server.
Lack of Flow Control With UDP (Continued)

```c
#include "unp.h"

static void recvfrom_int(int);
static int count;

void
dg_echo(int sockfd, SA *cliaddr, socklen_t clilen)
{
    socklen_t len;
    char   mesg[MAXLINE];
    Signal(SIGINT, recvfrom_int);
```
Lack of Flow Control With UDP (Continued)

```c
for ( ; ; ) {
    len = clilen;
    Recvfrom(sockfd, msg, MAXLINE, 0, pcliaddr, &len);
    count++;
}
```

```c
static void
recvfrom_int(int signo)
{
    printf("\nreceived %d datagrams\n", count);
    exit(0);
}
```

---

**Figure 8.20**  *dg_echo* function that counts received datagrams.
Lack of Flow Control With UDP (Continued)

```
bsdi % netstat -s | tail
udp: 80300 datagrams received
  0 with incomplete header
  0 with bad data length field
  0 with bad checksum
  12 dropped due to no socket
77725 broadcast/multicast datagrams dropped due to no socket
1970 dropped due to full socket buffers
  593 delivered
  70592 datagrams output
```

```
bsdi % udpserv06

start our server
we run the client here
type our interrupt key after client is finished

received 82 datagrams
```

```
bsdi % netstat -s | tail
udp: 82294 datagrams received
  0 with incomplete header
  0 with bad data length field
  0 with bad checksum
  12 dropped due to no socket
77725 broadcast/multicast datagrams dropped due to no socket
3882 dropped due to full socket buffers
  675 delivered
  70592 datagrams output
```

Figure 8.21 Output on server host.
Lack of Flow Control With UDP (Continued) – UDP Socket Receive Buffer

- 3882 – 1970 = 1912 dropped due to full socket buffer
- 82 datagrams received => 1994 datagrams
- 6 datagrams were never received

- Repeat this test several times and receive: 37, 108, 30, 108, 114

- We can change the socket’s receive buffer using SO_RCVBUF socket option
- Default size in UDP for a BSD/OS is 41,600 bytes => room for only
  41,600 = 29 of our datagrams
  1400 Can change to 240 K bytes size
- Now we receive 115, 168, 179, 145, and 133 in Stevens example runs
- Not much bigger than 37, 108, 30, 108, 114 from before buffer size change
Lack of Flow Control With UDP (Continued) –
UDP Socket Receive Buffer

```c
#include "unp.h"

static void recvfrom_int(int);
static int count;

void
dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
    int n;
    socklen_t len;
    char mesg[MAXLINE];

    Signal(SIGINT, recvfrom_int);

    n = 240 * 1024;
    Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

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

        count++;
    }
}

static void
recvfrom_int(int signo)
{
    printf("\nreceived %d datagrams\n", count);
    exit(0);
}
```

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