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.
Figure 8.1 Socket functions for UDP client–server.
recvfrom and sendto functions

sockfd
*buff
nbytes
flags

Descriptor.
Pointer to buffer to read or write.
Number of Bytes to read or write.
Set to 0 for now

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).

sendto

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)
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

[ from p. 240 ]
UDP echo Server

Redo echo client - server from chapter 5 but using UDP
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.
#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: dg_echo Function

Figure 8.4 shows the dg_echo function.

```c
#include "unp.h"

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

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

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

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.
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.

![Diagram of UDP client-server with two clients](image)

Figure 8.6 Summary of UDP client-server with two clients.
**UDP echo Client Main Function**

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**
```c
#include "unp.h"

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

    if (argc != 2)
        err_quit("usage: udpcli <IPaddress>\n```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```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.
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 got.
```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.
#include "unp.h"

static void recvfrom_int(int);
static int count;

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

    Signal(SIGINT, recvfrom_int);
for (; ; ) {
    len = clilen;
    Recvfrom(sockfd, msg, MAXLINE, 0, pcliaddr, &len);
    count++;
}

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

Figure 8.20  dg_echo function that counts received datagrams.
bsdhi % 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

bsdhi % udpserv06

received 82 datagrams

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

type

bsdhi % 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.
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.
```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.