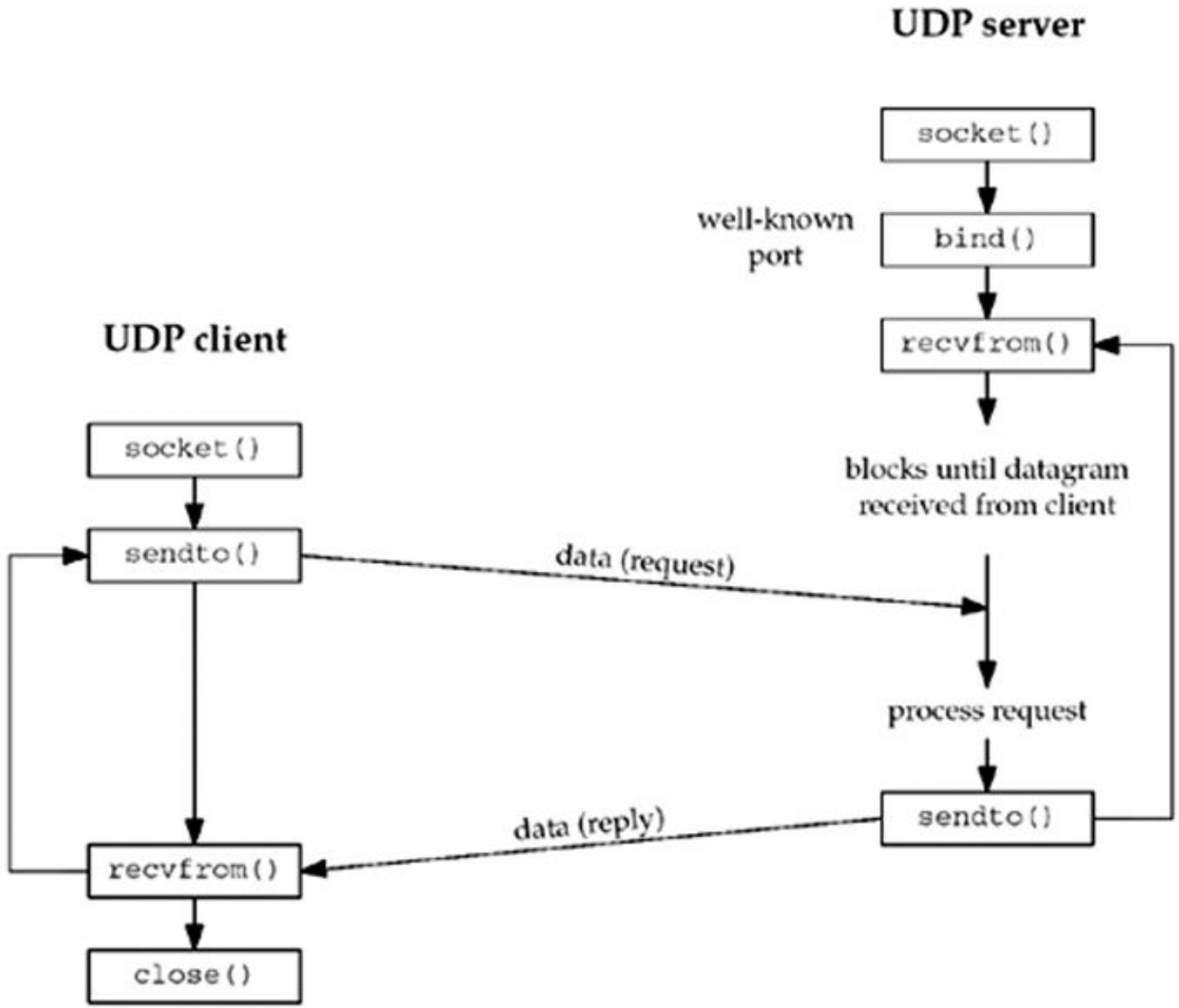
Unit –V
Elementary UDP SOCKETS

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.
- 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, 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.



'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);
```

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

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.***
- 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).**

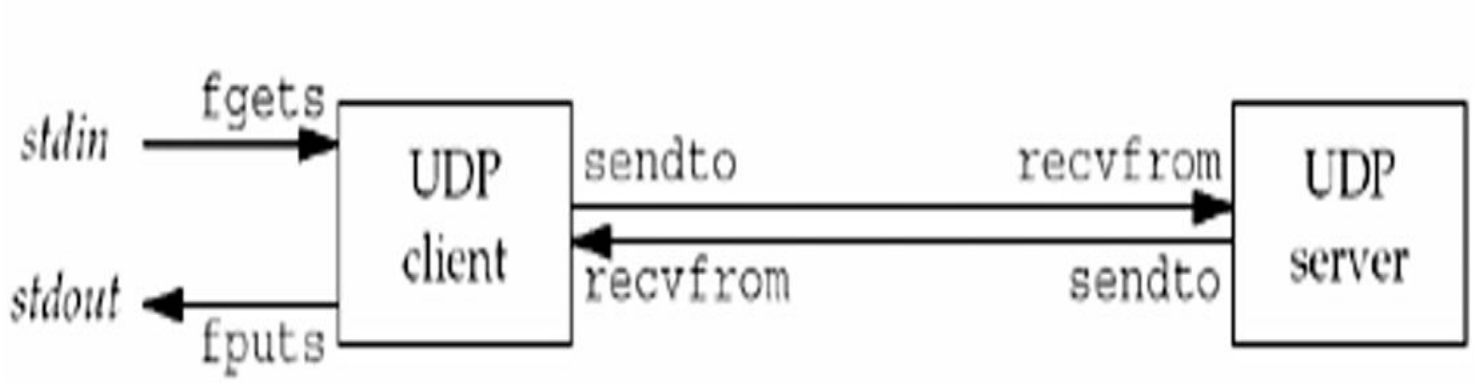
- The final two arguments to recvfrom are similar to the final two arguments to **accept:** The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP).
- The final two arguments to sendto are similar 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.**

UDP Echo Server: 'main' Function

Figure 8.2. Simple echo client/server using UDP.



```
 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.

UDP Echo Server: '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

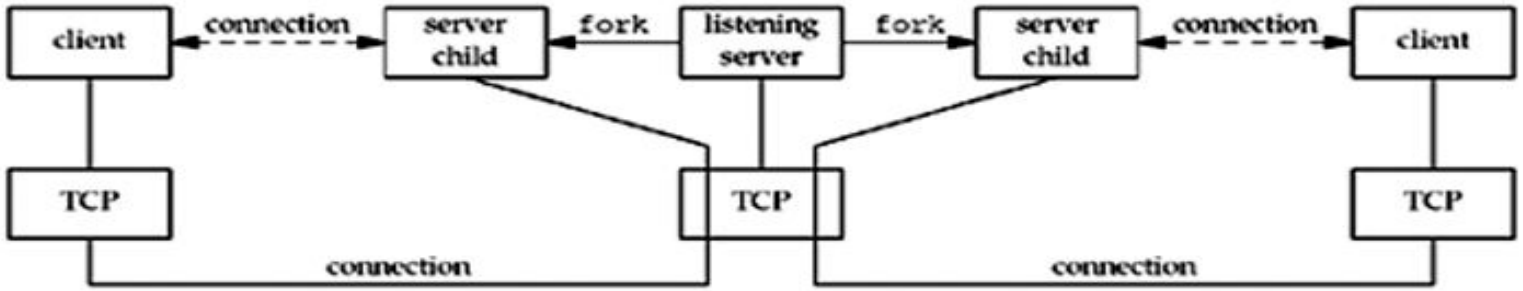
8–12 This function is a simple loop that reads the next datagram arriving at the server's port using `recvfrom` and sends it back using `sendto`.

- 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.

- 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.

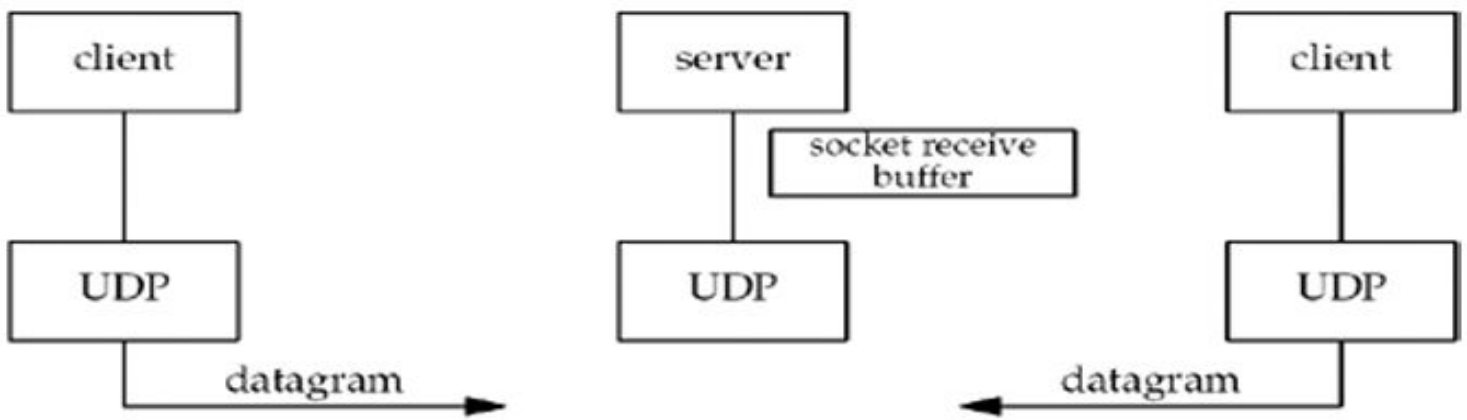
Figure 8.5. Summary of TCP client/server with two clients.



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.



UDP Echo Client: 'main' Function

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.

UDP Echo Client: 'dg_cli' Function

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.
- 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.