Elementary UDP Sockets

Sanjaya Kumar Jena

Functions of Transport Layer

The **transport layer** is responsible for **process-to-process delivery** of the entire message. A process is an application program running on a host.

Other responsibilities:

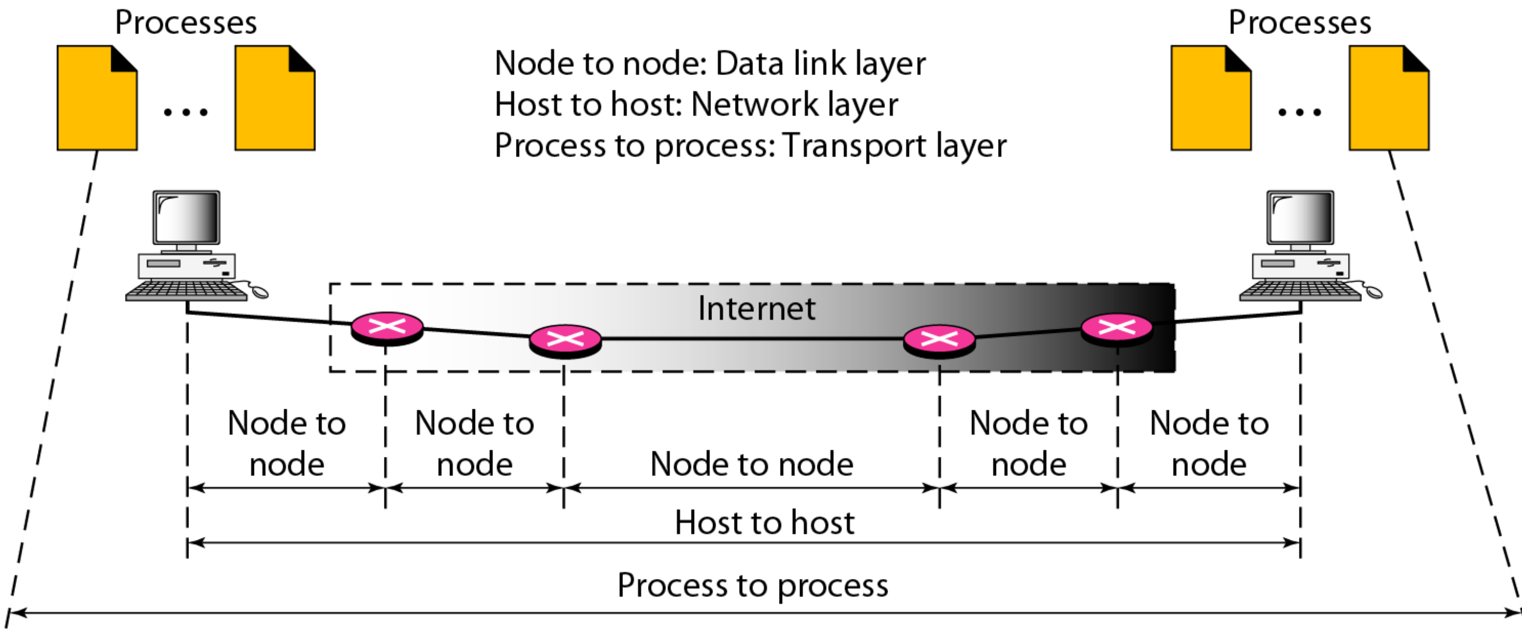
- Port addressing or service-point addressing
- Segmentation and reassembly
- Connection control
- Flow control
- Error control

Functions of Transport Layer

The **transport layer** is responsible for **process-to-process delivery** of the entire message. A process is an application program running on a host.

Other responsibilities:

- Port addressing or service-point addressing
- Segmentation and reassembly
- Connection control
- Flow control
- Error control



User Datagram Protocol (UDP)

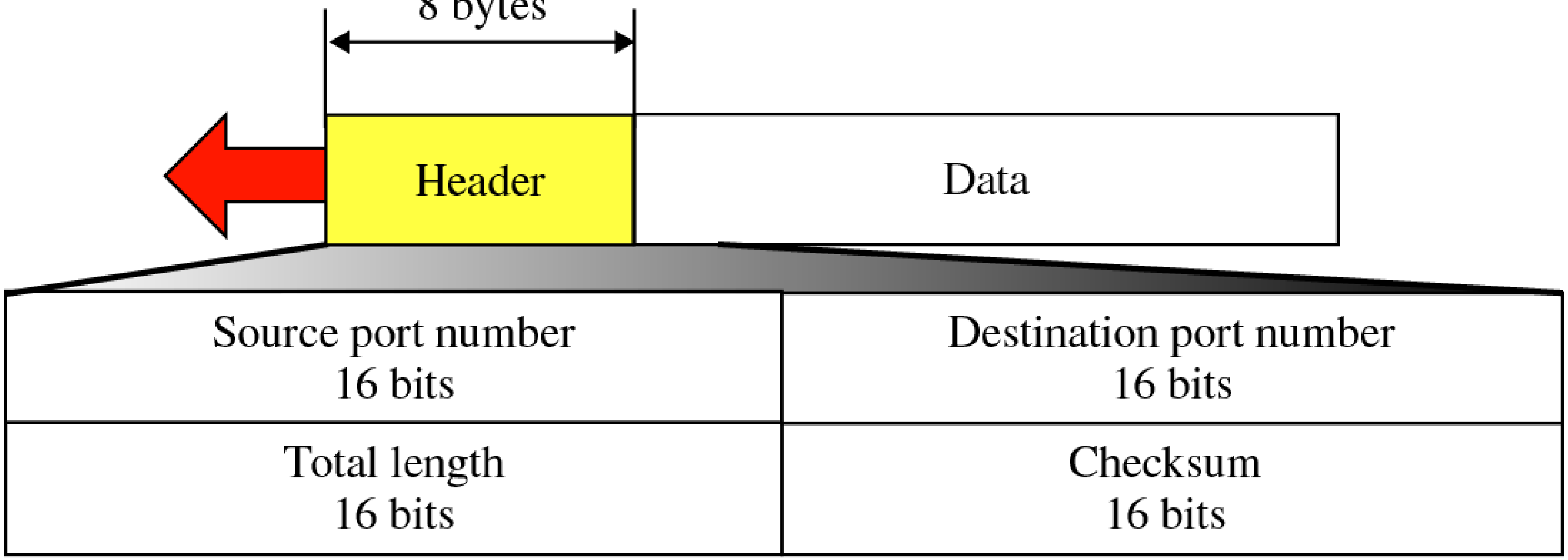
- UDP is a connectionless, unreliable transport protocol.
- UDP provides a connectionless service, as there need not be any longterm relationship between a UDP client and server. For example, a UDP client can create a socket and send a datagram to a given server and then immediately send another datagram on the same socket to a different server. Similarly, a UDP server can receive several datagrams on a single UDP socket, each from a different client.
- Each UDP datagram has a length. The length of a datagram is passed to the receiving application along with the data.
- Each UDP datagram has a length. The length of a datagram is passed to the receiving application along with the data.
- UDP can use either IPV4 or IPV6.
- In general, most TCP servers are concurrent and most UDP servers are iterative.

Well-known ports used with UDP

Port	Protocol	Description
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Nameserver	Domain Name Service
67	BOOTPs	Server port to download bootstrap information
68	BOOTPc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

User Datagram

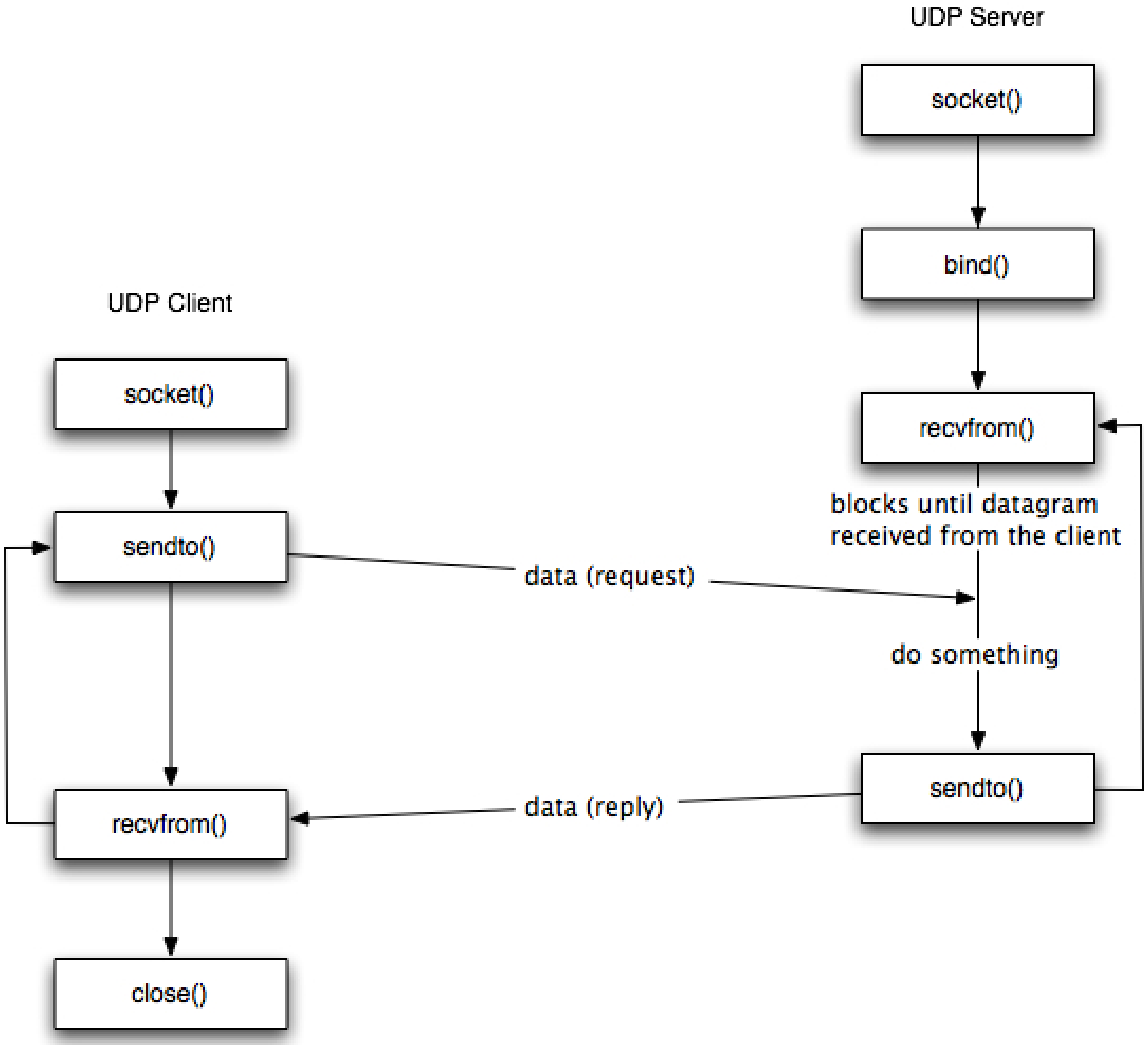
UDP packets called **user datagram**. Figure shows the format of a user datagram;



Length:: 16-bit field defines the total length of the user datagram (i.e. header + data)

Checksum:: Used to detect error over the entire user datagram.

UDP Client-Server Interaction



recvfrom and sendto functions

```
#include<sys/types.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;
```

sendto Examples

To send an integer

```
int num=10; len=sizeof(struct sockaddr_in);
sendto(sockfd,&num,sizeof(int), 0 ,
                (struct sockaddr *)&clientaddr,len);
```

To send a line of text

```
sendto(sockfd,"ITER-CSE-BTech\n",15, 0 ,
            (struct sockaddr *)&clientaddr,len);
```

To send a line of text

```
char sendline[1024]; fgets(sendline,1024,stdin);
sendto(sockfd,sendline,strlen(sendline), 0 ,
                      (struct sockaddr *)&clientaddr,&len);
```

recvfrom Examples

To receive an integer

```
int num; len=sizeof(struct sockaddr_in);
recvfrom(sockfd,&num,sizeof(int), 0 ,
                (struct sockaddr *)&clientaddr,&len);
printf("num recv=%d\n",num);
```

To receive a line of text

```
char recvline[1024];
ssize_t n;
n=recvfrom(sockfd,recvline,1024, 0 ,
             (struct sockaddr *)&clientaddr,len);
recvline[n]=0;
printf("string recvd=%s\n",recvline);
```

UDP Server program: userver.c

```
/*Socket : UDP Server*/
int main(int argc, char **argv){
  int sockfd,fd,len,p;
  struct sockaddr_in servaddr,clientaddr;
  char buff[1024];
  len=sizeof(struct sockaddr_in);
  sockfd=socket(AF_INET,SOCK_DGRAM,0);
  servaddr.sin_family=AF_INET;
  servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
  servaddr.sin_port=htons(0);
  bind(sockfd,(struct sockaddr *)&servaddr, sizeof(servaddr));
  getsockname(sockfd, (struct sockaddr *)&servaddr, &len);
  printf("After bind ephemeral port=%d\n",ntohs(servaddr.sin_port));
  recvfrom(sockfd,&p,4,0,(struct sockaddr *)&clientaddr,&len);
  printf("\nClient send::%d\n",p);
  printf("\nGive a string to send for client::");
  scanf("%s",buff);
  sendto(sockfd,buff,50,0,(struct sockaddr *)&clientaddr,len);
  close(sockfd);
}
```

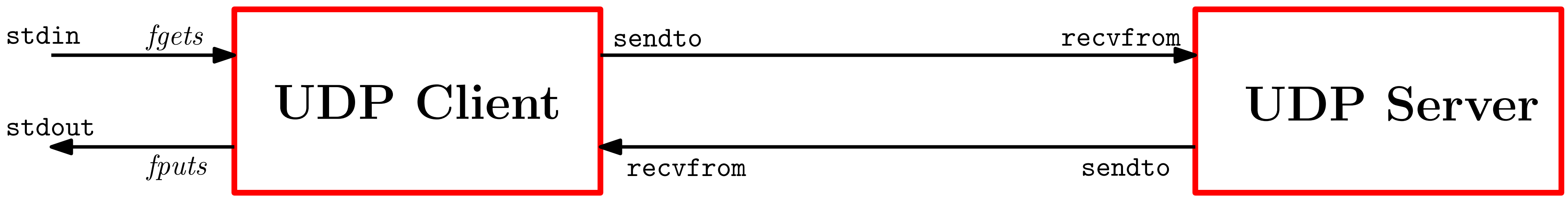
UDP Client program: uclient.c

```c
/*Socket : UDP Client*/
int main(int argc, char *argv[]){
 int sockfd,n,len;
 char recvline[1024];
 struct sockaddr_in servaddr;
 sockfd=socket(AF_INET,SOCK_DGRAM,0);
 servaddr.sin_family=AF_INET;
 servaddr.sin_addr.s_addr=inet_addr("127.0.0.1");
 servaddr.sin_port=htons(atoi(argv[1]));
 len=sizeof(servaddr);
 printf("Give a numbr for server::");
 scanf("%d",&n);
 sendto(sockfd,&n,4,0,(struct sockaddr *)&servaddr,len);
 recvfrom(sockfd,recvline,50,0,(struct sockaddr *)&servaddr,&len);
 recvline[n]=0;
 printf("\nServer send::%s\n",recvline);
 return 0;
}
```

UDP Echo Server Design

A complete UDP client/server using the elementary functions in UDP socket. This is an example is an echo server that performs the following steps:

1. The client reads a line of text from its standard input and writes the line to the server.
2. The server reads the line from its network input and echoes the line back to the client.
3. The client reads the echoed line and prints it on its standard output.



At Client side

1. fgets-Read from standard input
2. sendto- Send datagram to server
3. recvfrom- Receive datagram from server
4. fputs- Display on the standard output

At Server side

1. recvfrom- Receive datagram from client
2. sendto- send datagram to client

UDP Echo Server: Main Function

```
int main(int argc, char **argv){
    int sockfd;
    struct sockaddr_in servaddr,cliaddr;
    socklen_t len=sizeof(struct sockaddr_in);
    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(0);
    bind(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
    getsockname(sockfd,(struct sockaddr *)&servaddr,&len);
    printf("Port for client=%d\n",ntohs(servaddr.sin_port));
    dg_echo(sockfd,(struct sockaddr *)&cliaddr,sizeof(cliaddr));
    return 0;
}
```

UDP Echo Server: **dg_echo** Function

```
void dg_echo(int sockfd, struct sockaddr *pcliaddr,socklen_t clilen)
{
  int n;
  socklen_t len;
  char msg[MAXLINE];
  for(; ; ){
        len=clilen;
        n=recvfrom(sockfd,msg,MAXLINE,0,pcliaddr,&len);
        msg[n]=0;
        printf("msg from client=%s\n",msg);
        sendto(sockfd,msg,n,0,pcliaddr,len);
  }
}
```

About **`dg_echo`** Function

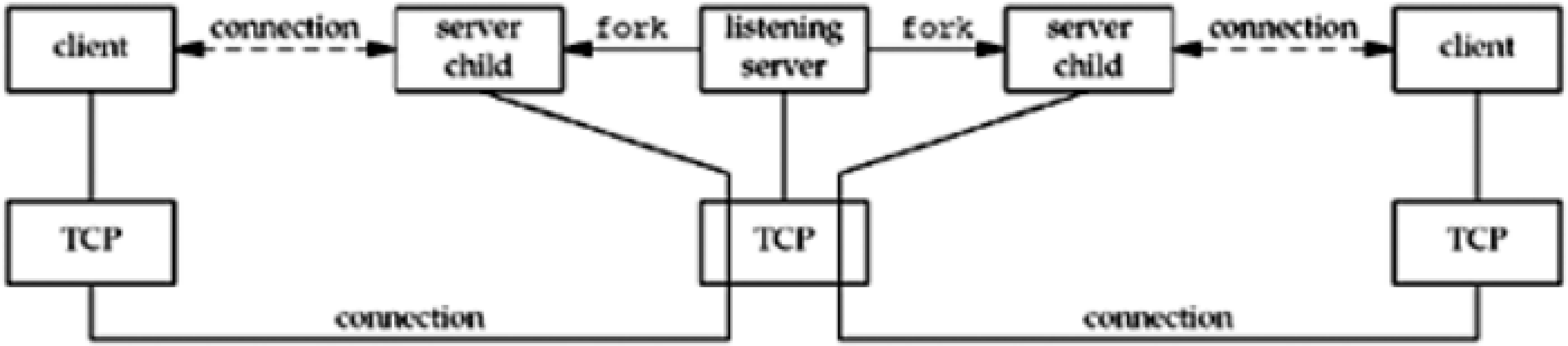
1. The function never terminates. Since UDP is a connectionless protocol, there is nothing like an **`EOF`** as we have with TCP.

2. 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.

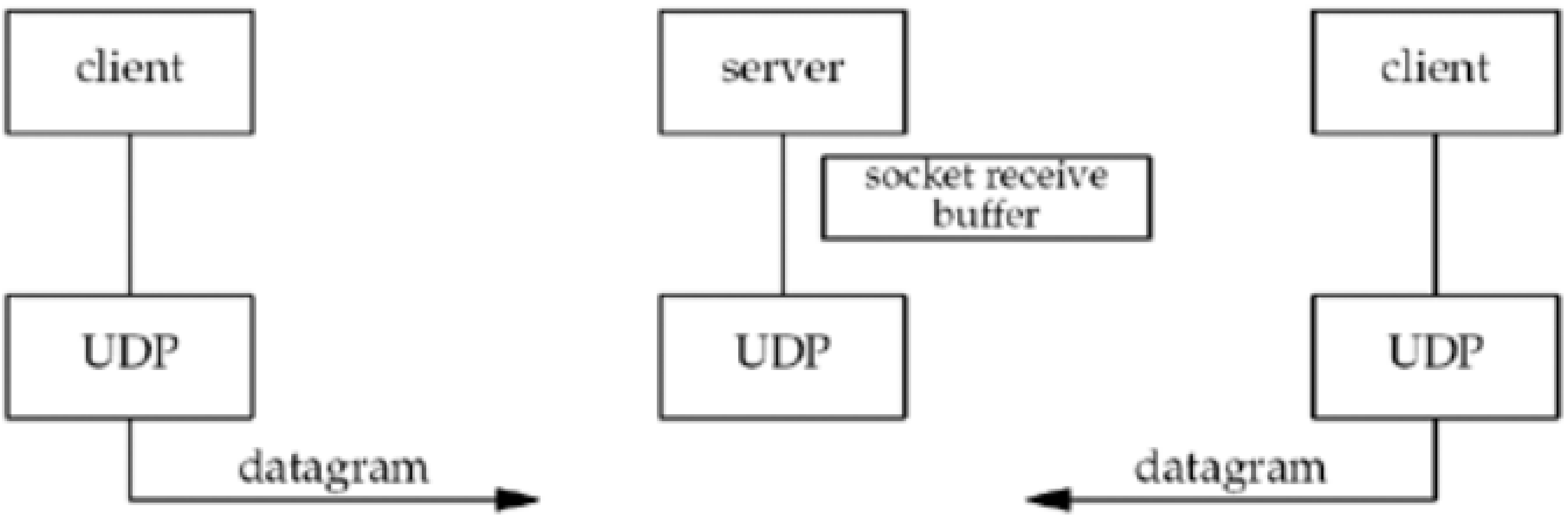
3. There is implied queuing taking place in the UDP layer for this socket. Indeed, each UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer.

4. When the process calls `recvfrom` , the next datagram from the buffer is returned to the process in a first-in, first-out (FIFO) order.

Summary of TCP client/server with two clients



Summary of UDP client/server with two clients



UDP Echo Client: Main Function

```
int main(int argc, char *argv[]){
    int sockfd;
    socklen_t len;
    struct sockaddr_in servaddr,cliaddr;
    if(argc!=3){
        fprintf(stderr,"Ushages %s <IP address>  <port>\n",argv[0]);
        return 1;
    }
    len=sizeof(struct sockaddr_in);
    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family=AF_INET;
    servaddr.sin_addr.s_addr=inet_addr(argv[1]);
    servaddr.sin_port=htons(atoi(argv[2]));
    sockfd=socket(AF_INET,SOCK_DGRAM,0);
    dg_cli(stdin,sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
    return 0;
}
```

UDP Echo Client: `dg_cli` Function

```
void dg_cli(FILE *fp, int sockfd,const struct sockaddr *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;
        fprintf(stderr,"\n\t\tServer reply:");
        fputs(recvline,stdout);
        fprintf(stderr,"\nEnter Message for server:");
    }
    printf("CTRL+D Pressed\n");
    sendto(sockfd,"",0, 0, pservaddr,servlen);
    n=recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);
    recvline[n]=0;
    fprintf(stderr,"\n\t\tServer reply:%ld byte length\n",strlen(recvline));
    fputs(recvline,stdout);
}
```

Observations on UDP Client/Server Interaction

- `bind` function in UDP client
- Case of lost datagrams
- Verifying received responses
- Server not running and asynchronous error
- `connect` function with UDP
- Calling `connect` multiple times for a UDP socket
- Lack of flow control with UDP
- Determining outgoing interface with UDP

bind function in client

- Normally, the client's IP address and port are chosen automatically by the kernel in the very first call to **sendto**.
- The client can call **bind** to choose ip address and port.
- If these two values (IP and Port) for the client are chosen by the kernel, the client's ephemeral port is chosen once, on the first **sendto**, and then it never changes.
- The client's IP address, however, can change for every UDP datagram that the client sends, assuming the client does not bind a specific IP address to the socket.
- If the client host is multihomed, the client could alternate between two destinations, one going out the datalink on the left, and the other going out the datalink on the right (Refer figure-1).
- In this worst-case scenario, the client's IP address, as chosen by the kernel based on the outgoing datalink, would change for every datagram.

Summary of UDP Example

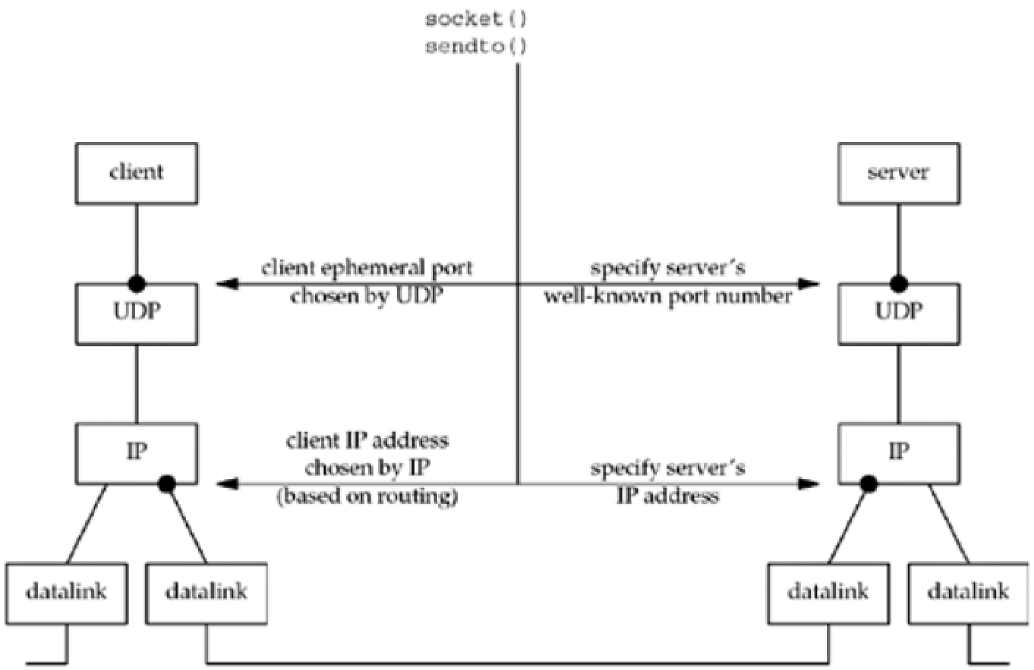


Figure 1: Client's perspective

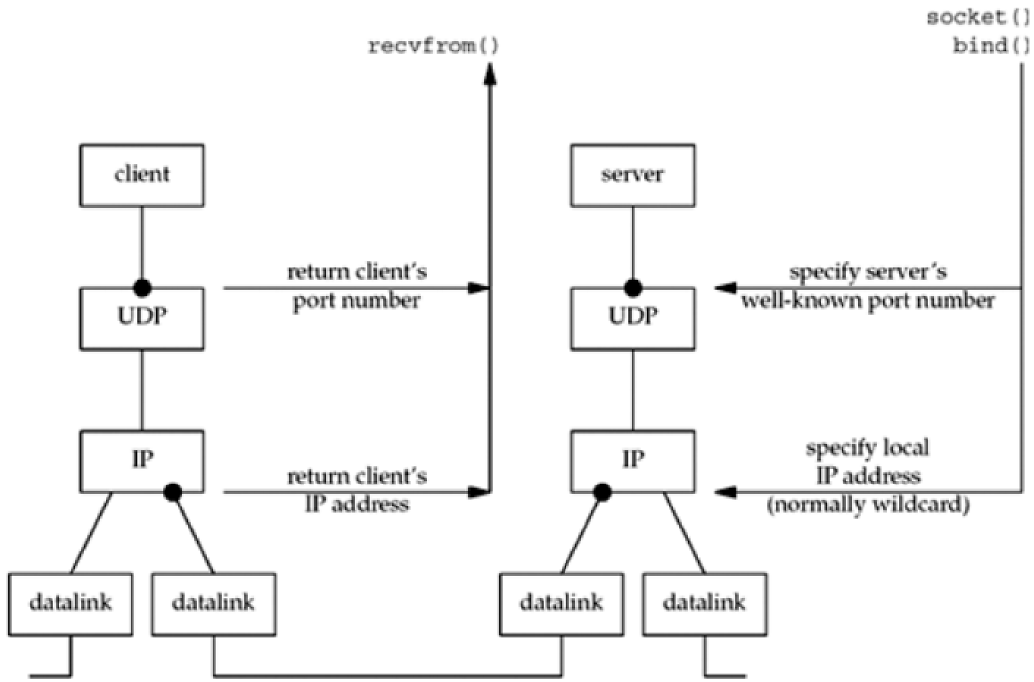


Figure 2: Server's perspective

Lost Datagrams

- Client datagram lost: the client will block forever in its call to **recvfrom** in the function **dg_cli**, waiting for a server reply that will never arrive.
- Server's reply is lost: if the client datagram arrives at the server but the server's reply is lost, the client will again block forever in its call to **recvfrom**.

Prevention

To prevent the lost datagram is to place a timeout on the client's call to **recvfrom**.

Different ways to place timeout

- **recvfrom** with a timeout using SIGALRM
- **recvfrom** with a timeout using **select**
- **recvfrom** with a timeout using the **SO_RECVTIMEO** socket option

Received Responses Verification

```
void dg_cli(FILE *fp, int sockfd,const struct sockaddr *pservaddr,
                                        socklen_t servlen)
{  ......
   struct sockaddr *preply_addr;
   preply_addr = malloc(servlen);
   while(fgets(sendline, MAXLINE,fp)!=NULL){
     sendto(sockfd,sendline,strlen(sendline), 0, pservaddr,servlen);
     len = servlen;
     n = recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
     if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0) {
            printf("reply from %s (ignored)\n",
                     inet_ntoa(((struct sockaddr_in *)preply_addr)->sin_addr));
            continue;
        }
        recvline[n]=0;
        fprintf(stderr,"\n\t\tServer reply:");
        fputs(recvline,stdout);
  }}
```

Server Not Running & Asynchrous Error

- Start the client without starting the server.
- Type in a single line to the client. The client blocks forever in its call to **`recvfrom`** , waiting for a server reply that will never appear.
- Check **`tcpdump`** output when server process not started on server host.
- **Result at `tcpdump`:** The client datagram sent but the server host responds with an ICMP `port unreachable` (i.e ICMP error).
- This ICMP error, however, is not returned to the client process.
- **Asynchronous error:** This ICMP error. The error was caused by **`sendto`**, but **`sendto`** returned successfully. A successful return from a UDP output operation only means there was room for the resulting IP datagram on the interface output queue.
- **The basic rule is that an asynchronous error is not returned for a UDP socket unless the socket has been connected.**

`connect` Function with UDP

- We observed that an asynchronous error is not returned on a UDP socket unless the socket has been connected. So, call **`connect`** for a UDP socket.
- **`connect`** for UDP socket does not result in anything like a TCP connection: There is no three-way handshake.
- Use of connect makes the kernel just checks for any immediate errors records the IP address and port number of the peer (from the socket address structure passed to connect), and returns immediately to the calling process.
- So, UDP socket; (i) An **unconnected** UDP socket, the default when we create a UDP socket, (ii) A **connected** UDP socket, the result of calling connect on a UDP socket.

connected UDP Socket

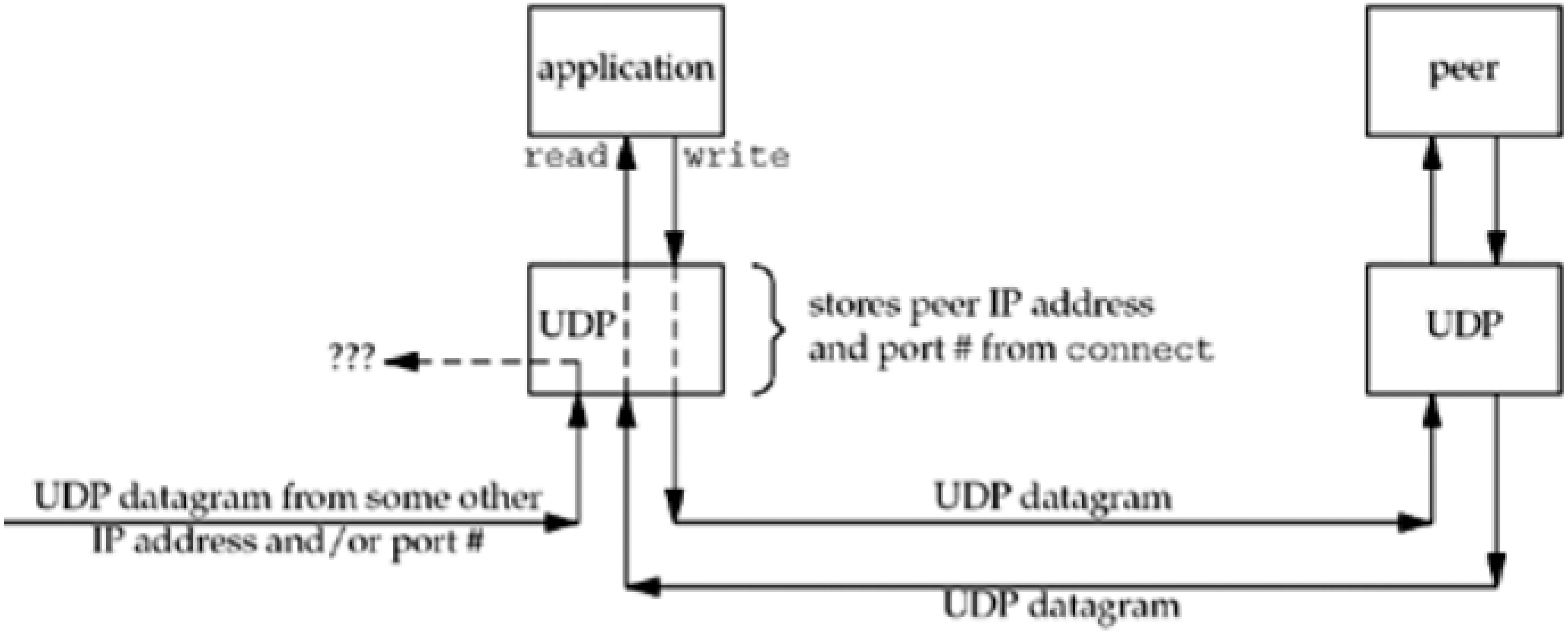


Figure 3: Connected UDP socket datagram send/receive

Changes with a connected UDP socket compared to the default unconnected UDP socket

- In place of **`sendto`**, use **`write`** or **`send`** function call.
- In place of **`recvfrom`**, use **`read`** or, **`recv`**, or **`recvmsg`** function call.
- Asynchronous errors are returned to the process for connected UDP sockets.
- We can use **`sendto`** for a connected UDP socket, but we cannot specify a destination address. The fifth argument `NULL` and the sixth argument 0.
- A **`connected`** UDP socket exchanges datagrams with only one IP address, because it is possible to connect to a multicast or broadcast address.

connected UDP socket Example

```
void dg_cli(FILE *fp, int sockfd,const struct sockaddr *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;
      fprintf(stderr,"\n\t\tServer reply:");
      fputs(recvline,stdout);
   }
}
```

Calling `connect` Multiple Times for a UDP Socket

A process with a connected UDP socket can call connect again for that socket for one of two reasons:

- To specify a new IP address and port
- To unconnect the socket - Call **`connect`** but set the family member of the socket address structure (sin_family for IPv4 or sin6_family for IPv6) to `AF_UNSPEC`.

Determination of Outgoing Interface with UDP

- A connected UDP socket can also be used to determine the outgoing interface that will be used to a particular destination.
- A side effect of the **`connect`** function when applied to a UDP socket: The kernel chooses the local IP address (assuming the process has not already called **`bind`** to explicitly assign this).
- This local IP address is chosen by searching the routing table for the destination IP address, and then using the primary IP address for the resulting interface.

UDP program that uses `connect` to determine outgoing interface

A simple UDP program that **`connects`** to a specified IP address and then calls **`getsockname`**, to print the local IP address and port.

```
int main(int argc, char *argv[]){
    :::::::::::
    sockfd=socket(AF_INET,SOCK_DGRAM,0);
    connect(sockfd, (struct sockaddr *)&servaddr, len);

    len=sizeof(cliaddr);
    getsockname(sockfd,(struct sockaddr *)&cliaddr,&len);

    printf("IP: %s\n",inet_ntoa(cliaddr.sin_addr));
    printf("Port: %d\n",ntohs(cliaddr.sin_port));
    return 0;
}
```

Lack of Flow Control with UDP

- The effect of UDP not having any flow control.
- Modify **`dg_cli`** function to send a fixed number of datagrams. It no longer reads from standard input. This function writes 2,000 1,400-byte UDP datagrams to the server.
- We next modify the server to receive datagrams and count the number received.
- This server no longer echoes datagrams back to the client.
- When we terminate the server with our terminal interrupt key (`SIGINT`), it prints the number of received datagrams and terminates.
- Examine using **`netstat -s`** on the server, both before and after running the client-server applications.

Example: Lack of Flow Control with UDP

dg_cli function in client that writes a fixed number of datagrams to the server.

```
#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[DGLEN];
   for (i = 0; i < NDG; i++) {
     sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
   }
}
```

Example: Lack of Flow Control with UDP

dg_echo function in in server that counts received datagrams.

```
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, mesg, MAXLINE, 0, pcliaddr, &len);
     count=count+1;
  }
}

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

Observations: Lack of Flow Control with UDP

- Carefully look at the **`netstat`** output about the total number of packets received at the server before and after the code run.
- Draw the conclusion about the datagrams are lost or not based on counter value.
- The counter indicates how many datagrams were received by UDP but were discarded because the receiving socket's receive queue was full
- The number of datagrams received by the server in this example is not predictable. It depends on many factors, such as the network load, the processing load on the client host, and the processing load on the server host.

UDP Socket Receive Buffer

- The number of UDP datagrams that are queued by UDP for a given socket is limited by the size of that socket's receive buffer.
- The default size of the UDP socket receive buffer can be obtained as;

```
int n,optlen;
getsockopt(sockfd, SOL_SOCKET,SO_RCVBUF,&n, &optlen);
printf("%d\n",n);
```

- We can change this with the SO_RCVBUF socket option. Let sets the socket receive buffer to 240 KB

```
int n;
n=240*1024;
setsockopt(sockfd, SOL_SOCKET,SO_RCVBUF,&n, sizeof(int));
```

Single Server Model to Multiplex TCP and UDP Socket

- Combination of concurrent TCP echo server and iterative UDP echo server
- Use of **`select`** or **`poll`** to multiplex TCP and UDP socket
- Use of socket option `SO_REUSEADDR` for address reuse

THANK YOU