Networks lab
Experiment-3

Samudrala Avinash
B180409CS

Socket Programming:

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

Socket:

A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

Headers Used:

1. sys/socket.h
2. arpa/inet.h
3. unistd.h

Functions Used:

1. int **socket** (int *domain*, int *type*, int *protocol*); <sys/socket.h>: creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

   The *domain* argument specifies a communication domain; this selects the protocol family which will be used for communication. These families are defined in
   <sys/socket.h>.

   The *type* argument specifies the communication semantics.

   The *protocol* specifies a particular protocol to be used with the socket.
   On success, a file descriptor for the new socket is returned. On error, -1 is returned, and errno is set appropriately.

2. int **bind** (int *socket*, const struct sockaddr ***address*, socklen_t *address_len*); <sys/socket.h>: It shall assign a local socket address to a socket identified by descriptor socket that has no local socket address assigned. Sockets created with the socket() function are initially unnamed; they are identified only by their address family.

   The *socket* argument Specifies the file descriptor of the socket to be bound.

   The *address* argument points to a sockaddr structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.

   The *address_len* argument specifies the length of the sockaddr structure pointed to by the address argument.

   Upon successful completion, bind () shall return 0; otherwise, -1 shall be returned and errno set to indicate the error.

3. int **listen** (int *socket*, int *backlog*); <sys/socket.h>: The listen () function shall mark a connection-mode socket, specified by the *socket* argument, as accepting connections.

   The *backlog* argument defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error

   Upon successful completions, listen () shall return 0; otherwise, -1 shall be returned and errno set to indicate the error.

4. int **accept** (int *socket*, struct sockaddr * *address*, socklen_t * *address_len*); <sys/socket.h>: It shall extract the first connection on the queue of pending connections, create a new socket with the same socket type protocol and address family as the specified socket, and allocate a new file descriptor for that socket.

   The *socket* argument specifies a socket that was created with socket(), has been bound to an address with bind(), and has issued a successful call to listen(). The *address* argument is either a null pointer, or a pointer to a sockaddr structure where the address of the connecting socket shall be returned.

   The *address_len* argument points to a socklen_t structure which on input specifies the length of the supplied sockaddr structure, and on output specifies the length of the stored address.

   Upon successful completion, accept () shall return the non-negative file descriptor of the accepted socket. Otherwise, -1 shall be returned and errno set to indicate the error.

5. int **connect** (int *socket*, const struct sockaddr **address*, socklen_t *address_len*); <sys/socket.h>: The connect () function shall attempt to make a connection on a socket. The *socket* argument specifies the file descriptor associated with the socket.

   The *address* argument points to a sockaddr structure containing the peer address. The length and format of the address depend on the address family of the socket.

   The *address_len* argument specifies the length of the sockaddr structure pointed to by the address argument.

   Upon successful completion, connect () shall return 0; otherwise, -1 shall be returned and errno set to indicate the error.

6. ssize_t **send** (int *socket*, const void **buffer*, size_t *length*, int *flags*); <sys/socket.h>: The send () function shall initiate transmission of a message from the specified socket to its peer. The send () function shall send a message only when the socket is connected.

   The *socket* argument specifies the socket file descriptor.

   The *buffer* argument points to the buffer containing the message

   to send. The *length* argument specifies the length of the

   message in bytes.

   The *flags* argument specifies the type of message transmission.

   Upon successful completion, send () shall return the number of bytes sent. Otherwise, -1 shall be returned and errno set to indicate the error.

7. ssize_t **recv** (int *socket*, void **buffer*, size_t *length*, int *flags*); <sys/socket.h>: The recv () function shall receive a message from a connection-mode or connectionless-mode socket. It is normally used with connected sockets because it does not permit the application to retrieve the source address of received data.

   The *socket* argument specifies the socket file descriptor.

   The *buffer* argument points to a buffer where the message should be stored.

The *length* argument specifies the length in bytes of the buffer pointed to by the buffer argument.

The *flags* argument specifies the type of message reception.

Upon successful completion, recv () shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, recv () shall return 0. Otherwise, -1 shall be returned and errno set to indicate the error.

8. ssize_t **sendto** (int *socket*, const void **message*, size_t *length*, int *flags*, const struct sockaddr **dest_addr*, socklen_t *dest_len*); <sys/socket.h>: The sendto () function shall send a message through a connection-mode or connectionless-mode socket. If the *socket* is connectionless-mode, the message shall be sent to the address specified by *dest_addr*. If the socket is connection-mode, *dest_addr* shall be ignored.

   The *socket* argument specifies the socket file descriptor.

   The *buffer* argument points to the buffer containing the message

   to send. The *length* argument specifies the length of the

   message in bytes.

   The *flags* argument specifies the type of message transmission.

   The *dest_addr* argument points to a sockaddr structure containing the destination address. The length and format of the address depend on the address family of the socket.

   The *dest_len* argument specifies the length of the sockaddr structure pointed to by the dest_addr argument.
   Upon successful completion, sendto () shall return the number of bytes sent. Otherwise, -1 shall be returned and errno set to indicate the error.

9. ssize_t **recvfrom** (int *socket*, void *restrict *buffer*, size_t *length*, int *flags*, struct sockaddr
   *restrict *address*, socklen_t *restrict *address_len*); <sys/socket.h>: The recvfrom () function shall receive a message from a connection-mode or connectionless-mode *socket*. It is normally used with connectionless-mode sockets because it permits the application to retrieve the source address of received data.

   The *socket* argument specifies the socket file descriptor.

The *buffer* argument points to a buffer where the message should be stored.

The *length* argument specifies the length in bytes of the buffer pointed to by the buffer argument.

The *flags* argument specifies the type of message reception.

The *address* argument is a null pointer, or points to a sockaddr structure in which the sending address is to be stored. The length and format of the address depend on the address family of the socket.

The *address_len* argument specifies the length of the sockaddr structure pointed to by the address argument.

Upon successful completion, recvfrom () shall return the length of the message in bytes. If no messages are available to be received and the peer has performed an orderly shutdown, recvfrom () shall return 0. Otherwise, the function shall return -1 and set errno to indicate the error.

10. int **close** (int *fildes*); <unistd.h>: The close () function shall deallocate the file descriptor indicated by *fildes*. To deallocate means to make the file descriptor available for return by subsequent calls to open() or other functions that allocate file descriptors(Here we used socket () function). All outstanding record locks owned by the process on the file associated with the file descriptor shall be removed (that is, unlocked). Upon successful completion, 0 shall be returned; otherwise, -1 shall be returned and errno set to indicate the error.

11. uint16_t **htons** (uint16_t *hostshort*); <arpa/inet.h>: convert 16-bit host byte order to network byte order. Returns the argument value converted from host to network byte order.

12. int **inet_pton** (int *af*, const char **src*, void **dst*); <arpa/inet.h>: The inet_pton () function converts an address in its standard text presentation form into its numeric binary form. The *af* argument specifies the family of the address. The AF_INET and AF_INET6 address families are supported.

    The *src* argument points to the string being passed in.
    The *dst* argument points to a buffer into which the function stores the numeric address; this must be large enough to hold the numeric address (32 bits for AF_INET, 128 bits for AF_INET6).

    Returns 1 if the conversion succeeds, with the address pointed to by dst in network byte order. It returns 0 if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string, or -1 with errno set to [EAFNOSUPPORT] if the af argument is unknown.

Commands used:

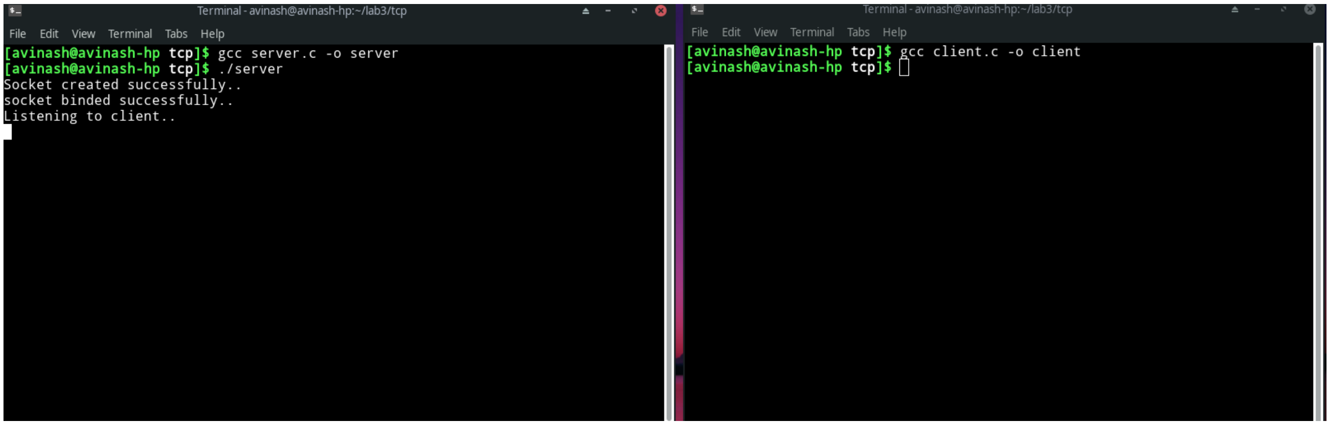
```
gcc server.c -o server && ./server
gcc client.c -o client && ./client
```

Process:

1. **TCP Connection**: In the server we first call the socket function to get the socket descriptor. Then we bind the socket to an address and port. The server starts listening when the listen function is called. Then it will accept the first connection waiting in the pending connection in the queue and starts communicating with the client. In the client we call the socket function to get the socket descriptor and connect to the server using connect function.The port number for the server is 8080.

2. **UDP Connection**: In the server and client we first call the socket function to get the socket descriptor. Then we bind the server's socket to an address and port. sendto and recvfrom functions are used for communication. Since UDP is connectionless, the server starts listening.The port number for the server is 8080.

Screenshots :

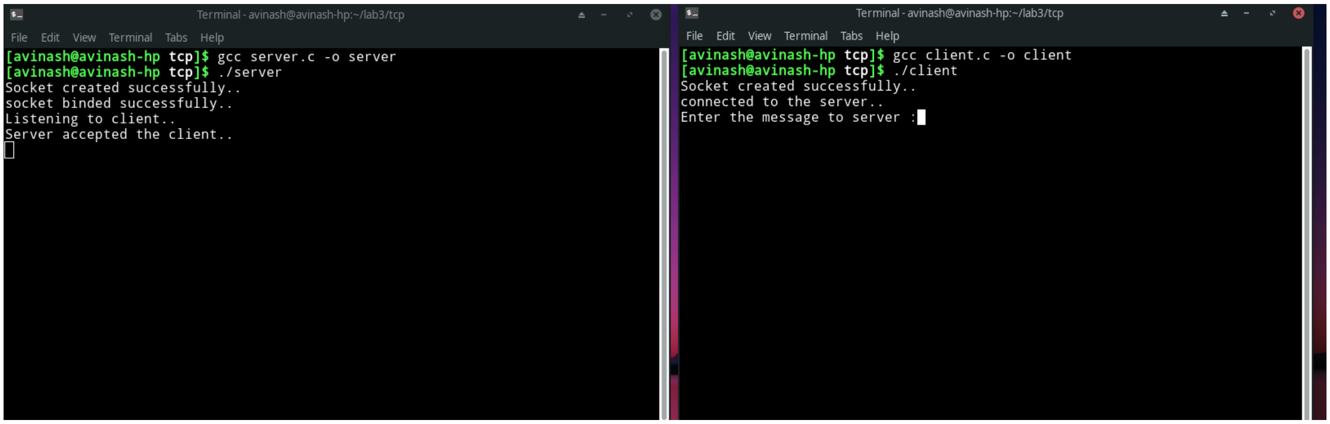

Terminal - avinash@avinash-hp:~/lab3/tcp
File Edit View Terminal Tabs Help

```
[avinash@avinash-hp tcp]$ gcc server.c -o server
[avinash@avinash-hp tcp]$ ./server
Socket created successfully..
socket binded successfully..
Listening to client..
```

Terminal - avinash@avinash-hp:~/lab3/tcp
File Edit View Terminal Tabs Help

```
[avinash@avinash-hp tcp]$ gcc client.c -o client
[avinash@avinash-hp tcp]$ 
```

Fig.1 : TCP Server-Client Connection. Started running the server.

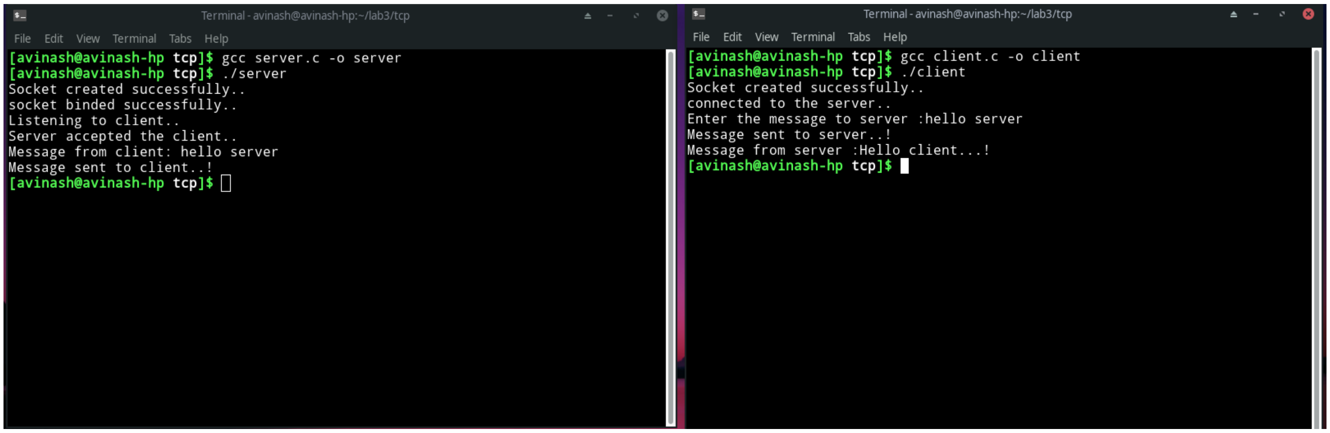


Terminal - avinash@avinash-hp:~/lab3/tcp
File Edit View Terminal Tabs Help

```
[avinash@avinash-hp tcp]$ gcc server.c -o server
[avinash@avinash-hp tcp]$ ./server
Socket created successfully..
socket binded successfully..
Listening to client..
Server accepted the client..
```

```
[avinash@avinash-hp tcp]$ gcc client.c -o client
[avinash@avinash-hp tcp]$ ./client
Socket created successfully..
connected to the server..
Enter the message to server :
```

Fig.2 : TCP Server-Client Connection. Started running the client.And client is waiting for user input.

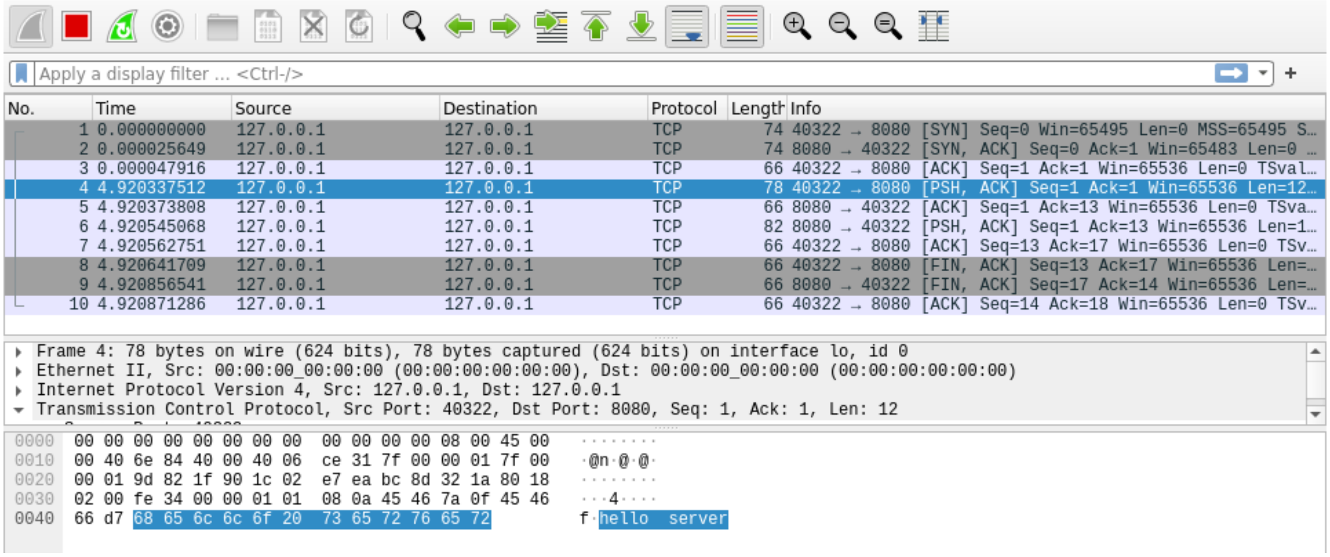


Terminal - avinash@avinash-hp:~/lab3/tcp
File Edit View Terminal Tabs Help

```
[avinash@avinash-hp tcp]$ gcc server.c -o server
[avinash@avinash-hp tcp]$ ./server
Socket created successfully..
socket binded successfully..
Listening to client..
Server accepted the client..
Message from client: hello server
Message sent to client..!
[avinash@avinash-hp tcp]$
```

```
[avinash@avinash-hp tcp]$ gcc client.c -o client
[avinash@avinash-hp tcp]$ ./client
Socket created successfully..
connected to the server..
Enter the message to server :hello server
Message sent to server..!
Message from server :Hello client...!
[avinash@avinash-hp tcp]$
```

Fig.3: TCP Server-Client Connection.User input was sent from client. “Hello client..!” was sent from the server. The connection was closed at the end.

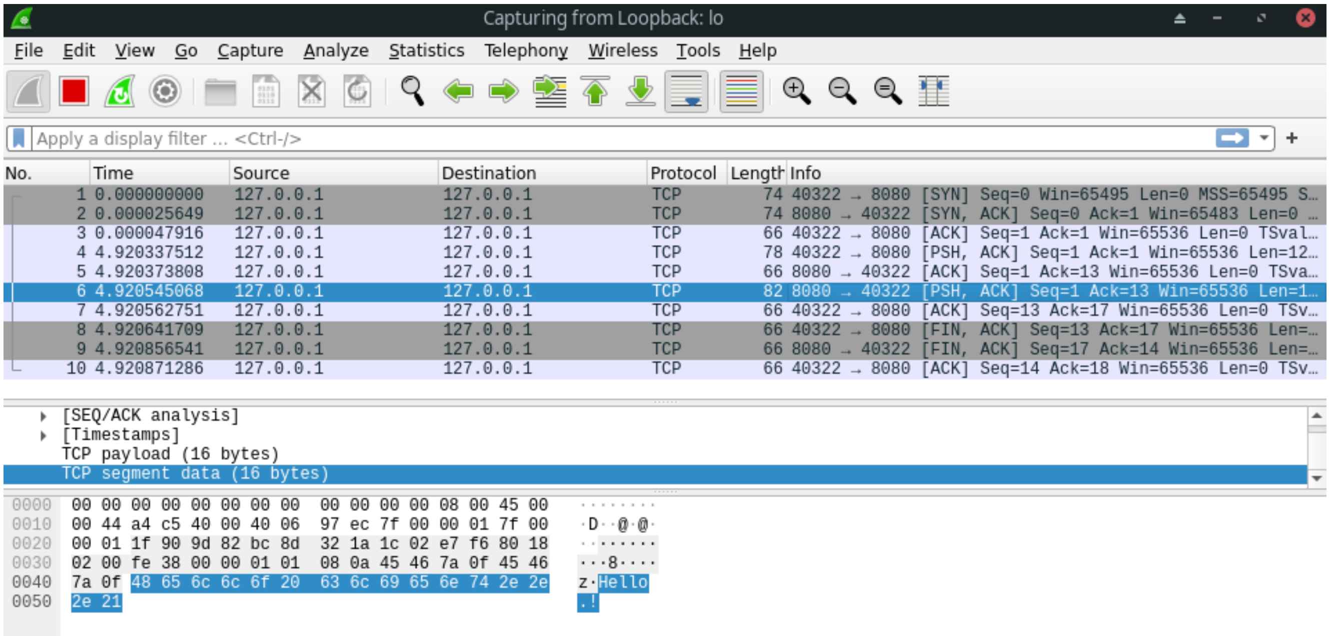


```
No.  Time         Source       Destination  Protocol Length Info
   1 0.000000000  127.0.0.1    127.0.0.1    TCP      74 40322 → 8080 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 S…
   2 0.000025649  127.0.0.1    127.0.0.1    TCP      74 8080 → 40322 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 …
   3 0.000047916  127.0.0.1    127.0.0.1    TCP      66 40322 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval…
   4 4.920337512  127.0.0.1    127.0.0.1    TCP      78 40322 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12…
   5 4.920373808  127.0.0.1    127.0.0.1    TCP      66 8080 → 40322 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSva…
   6 4.920545068  127.0.0.1    127.0.0.1    TCP      82 8080 → 40322 [PSH, ACK] Seq=1 Ack=13 Win=65536 Len=1…
   7 4.920562751  127.0.0.1    127.0.0.1    TCP      66 40322 → 8080 [ACK] Seq=13 Ack=17 Win=65536 Len=0 TSv…
   8 4.920641709  127.0.0.1    127.0.0.1    TCP      66 40322 → 8080 [FIN, ACK] Seq=13 Ack=17 Win=65536 Len=…
   9 4.920856541  127.0.0.1    127.0.0.1    TCP      66 8080 → 40322 [FIN, ACK] Seq=17 Ack=14 Win=65536 Len=…
  10 4.920871286  127.0.0.1    127.0.0.1    TCP      66 40322 → 8080 [ACK] Seq=14 Ack=18 Win=65536 Len=0 TSv…
```

```
▸ Frame 4: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▾ Transmission Control Protocol, Src Port: 40322, Dst Port: 8080, Seq: 1, Ack: 1, Len: 12
```

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········
0010  00 40 6e 84 40 00 40 06  ce 31 7f 00 00 01 7f 00   ·@n·@·@·
0020  00 01 9d 82 1f 90 1c 02  e7 ea bc 8d 32 1a 80 18   ········
0030  02 00 fe 34 00 00 01 01  08 0a 45 46 7a 0f 45 46   ···4····
0040  66 d7 68 65 6c 6c 6f 20  73 65 72 76 65 72         f·hello  server
```

Fig.4 : TCP Server-Client Connection. TCP Packet shown in wireshark when client sent the message to server. The highlighted part in bottom window shows the data sent.

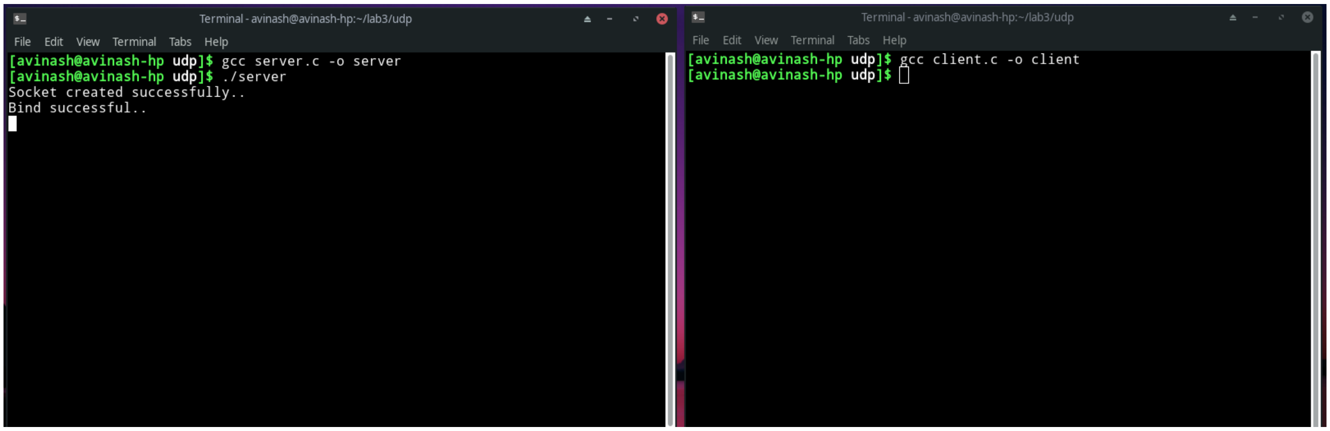

Capturing from Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.   Time         Source      Destination  Protocol Length Info
    1 0.000000000  127.0.0.1   127.0.0.1    TCP      74 40322 → 8080 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 S…
    2 0.000025649  127.0.0.1   127.0.0.1    TCP      74 8080 → 40322 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 …
    3 0.000047916  127.0.0.1   127.0.0.1    TCP      66 40322 → 8080 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval…
    4 4.920337512  127.0.0.1   127.0.0.1    TCP      78 40322 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12…
    5 4.920373808  127.0.0.1   127.0.0.1    TCP      66 8080 → 40322 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSva…
    6 4.920545068  127.0.0.1   127.0.0.1    TCP      82 8080 → 40322 [PSH, ACK] Seq=1 Ack=13 Win=65536 Len=1…
    7 4.920562751  127.0.0.1   127.0.0.1    TCP      66 40322 → 8080 [ACK] Seq=13 Ack=17 Win=65536 Len=0 TSv…
    8 4.920641709  127.0.0.1   127.0.0.1    TCP      66 40322 → 8080 [FIN, ACK] Seq=13 Ack=17 Win=65536 Len=…
    9 4.920856541  127.0.0.1   127.0.0.1    TCP      66 8080 → 40322 [FIN, ACK] Seq=17 Ack=14 Win=65536 Len=…
   10 4.920871286  127.0.0.1   127.0.0.1    TCP      66 40322 → 8080 [ACK] Seq=14 Ack=18 Win=65536 Len=0 TSv…

  ▸ [SEQ/ACK analysis]
  ▸ [Timestamps]
    TCP payload (16 bytes)
    TCP segment data (16 bytes)

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········
0010  00 44 a4 c5 40 00 40 06  97 ec 7f 00 00 01 7f 00   ·D··@·@·
0020  00 01 1f 90 9d 82 bc 8d  32 1a 1c 02 e7 f6 80 18   ········
0030  02 00 fe 38 00 00 01 01  08 0a 45 46 7a 0f 45 46   ···8····
0040  7a 0f 48 65 6c 6c 6f 20  63 6c 69 65 6e 74 2e 2e   z·Hello
0050  2e 21                                              .!
```

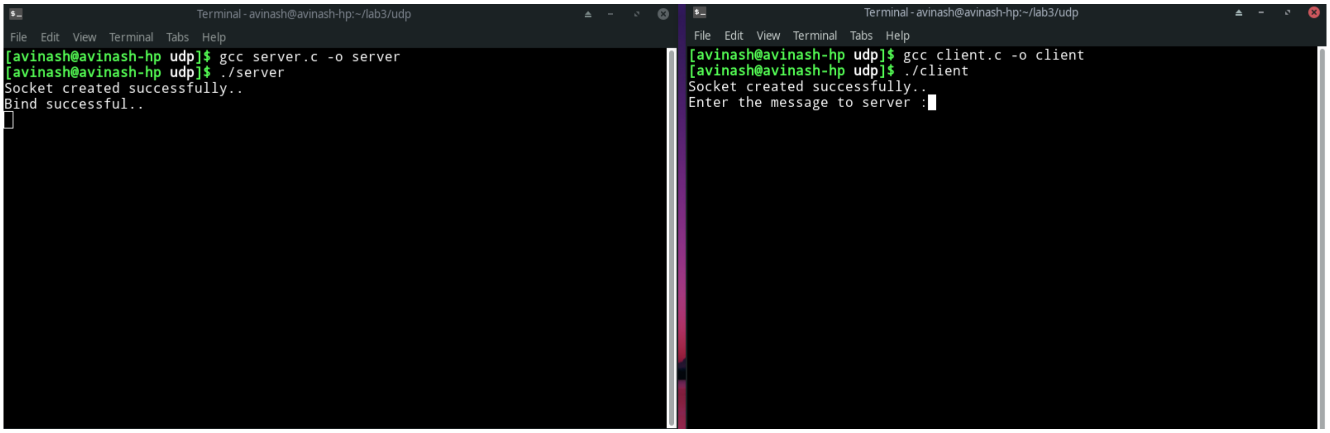
Fig.5 : TCP Server-Client Connection. TCP Packet shown in wireshark when server sent the message to client. The highlighted part in bottom window shows the data sent.


Terminal - avinash@avinash-hp:~/lab3/udp
File Edit View Terminal Tabs Help
Terminal - avinash@avinash-hp:~/lab3/udp
File Edit View Terminal Tabs Help

```
[avinash@avinash-hp udp]$ gcc server.c -o server
[avinash@avinash-hp udp]$ ./server
Socket created successfully..
Bind successful..
```

```
[avinash@avinash-hp udp]$ gcc client.c -o client
[avinash@avinash-hp udp]$
```

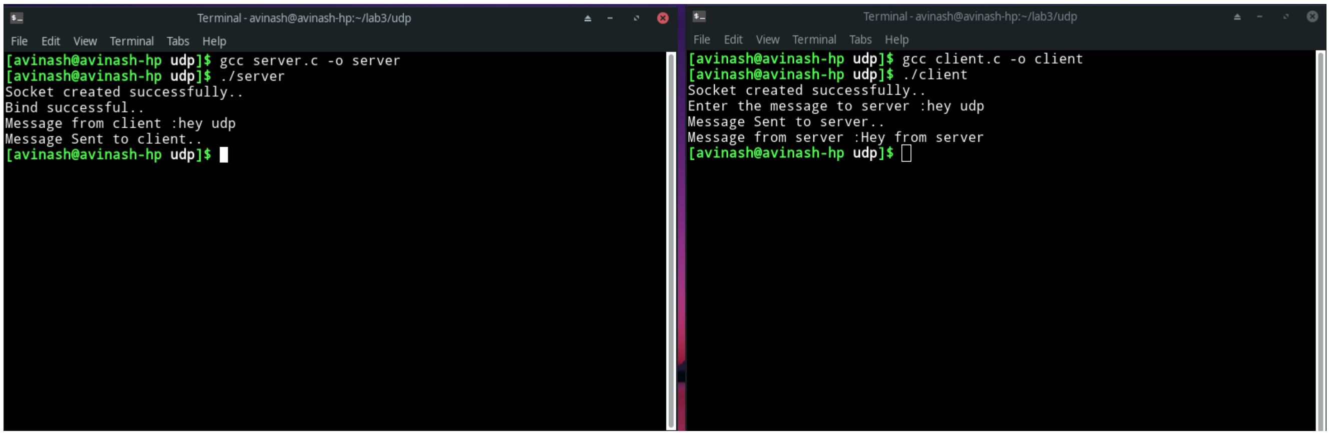
Fig.6 : UDP Server-Client Connection. Started running the server.


Terminal - avinash@avinash-hp:~/lab3/udp
File Edit View Terminal Tabs Help
Terminal - avinash@avinash-hp:~/lab3/udp
File Edit View Terminal Tabs Help

```
[avinash@avinash-hp udp]$ gcc server.c -o server
[avinash@avinash-hp udp]$ ./server
Socket created successfully..
Bind successful..
```

```
[avinash@avinash-hp udp]$ gcc client.c -o client
[avinash@avinash-hp udp]$ ./client
Socket created successfully..
Enter the message to server :
```

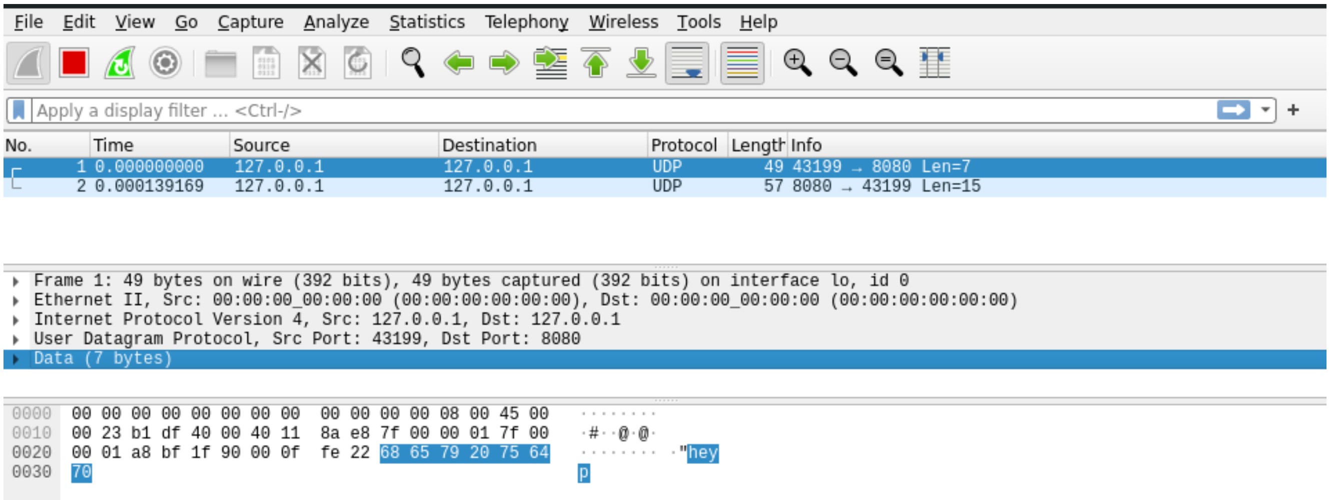
Fig.7 : UDP Server-Client Connection. Started running the client.Client waiting for user input.


Terminal - avinash@avinash-hp:~/lab3/udp
File Edit View Terminal Tabs Help
Terminal - avinash@avinash-hp:~/lab3/udp
File Edit View Terminal Tabs Help

```
[avinash@avinash-hp udp]$ gcc server.c -o server
[avinash@avinash-hp udp]$ ./server
Socket created successfully..
Bind successful..
Message from client :hey udp
Message Sent to client..
[avinash@avinash-hp udp]$
```

```
[avinash@avinash-hp udp]$ gcc client.c -o client
[avinash@avinash-hp udp]$ ./client
Socket created successfully..
Enter the message to server :hey udp
Message Sent to server..
Message from server :Hey from server
[avinash@avinash-hp udp]$
```

Fig.8: UDP Server-Client Connection.User input was sent from client. And “Hello client..!” was sent from the server.

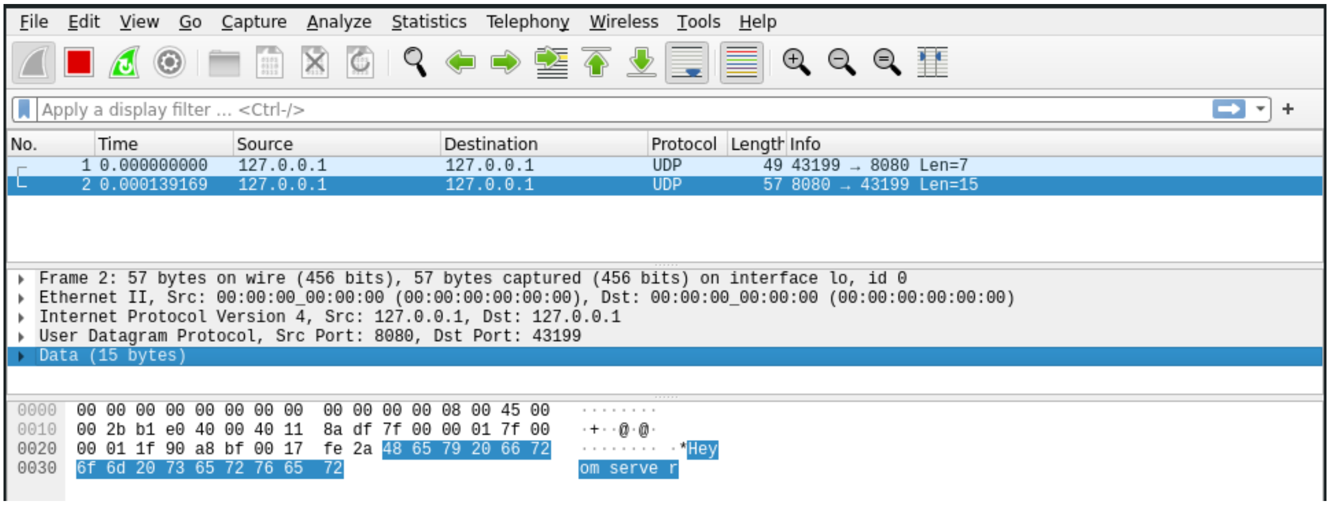

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.   Time          Source      Destination  Protocol  Length Info
    1 0.000000000   127.0.0.1   127.0.0.1    UDP          49 43199 → 8080 Len=7
    2 0.000139169   127.0.0.1   127.0.0.1    UDP          57 8080 → 43199 Len=15

Frame 1: 49 bytes on wire (392 bits), 49 bytes captured (392 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 43199, Dst Port: 8080
Data (7 bytes)

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········
0010  00 23 b1 df 40 00 40 11  8a e8 7f 00 00 01 7f 00   ·#··@·@·
0020  00 01 a8 bf 1f 90 00 0f  fe 22 68 65 79 20 75 64   ········ ·"hey
0030  70                                                 p
```

Fig.9 : UDP Server-Client Connection. UDP Packet shown in wireshark when client sent the message to server. The highlighted part in bottom window shows the data sent.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.   Time          Source      Destination  Protocol  Length Info
    1 0.000000000   127.0.0.1   127.0.0.1    UDP          49 43199 → 8080 Len=7
    2 0.000139169   127.0.0.1   127.0.0.1    UDP          57 8080 → 43199 Len=15

Frame 2: 57 bytes on wire (456 bits), 57 bytes captured (456 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 8080, Dst Port: 43199
Data (15 bytes)

0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········
0010  00 2b b1 e0 40 00 40 11  8a df 7f 00 00 01 7f 00   ·+··@·@·
0020  00 01 1f 90 a8 bf 00 17  fe 2a 48 65 79 20 66 72   ········ ·*Hey
0030  6f 6d 20 73 65 72 76 65  72                        om serve r
```

Fig.10 : UDP Server-Client Connection. UDP Packet shown in wireshark when server sent the message to client. The highlighted part in bottom window shows the data sent.