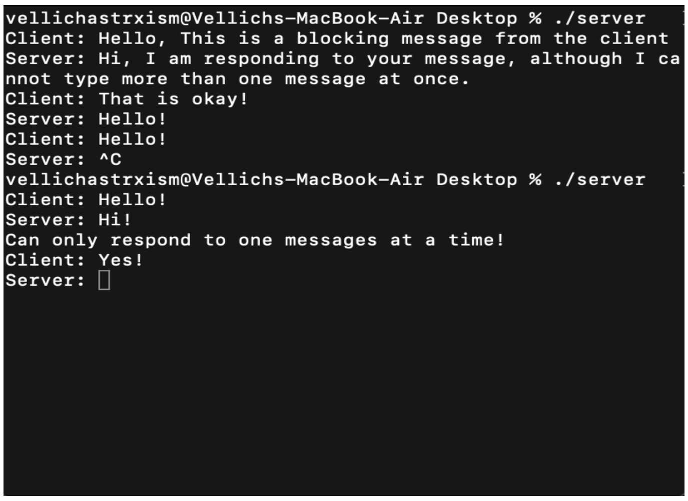
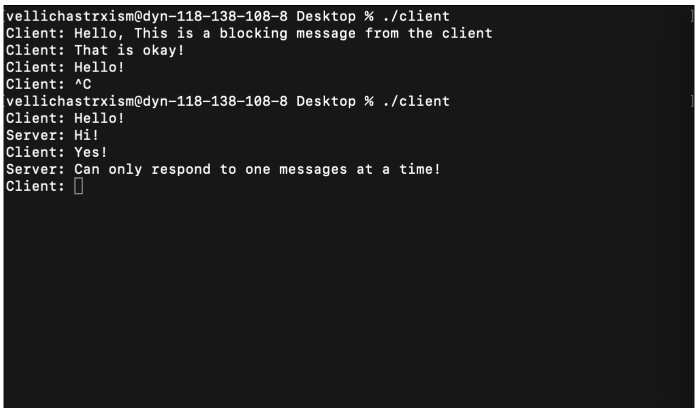
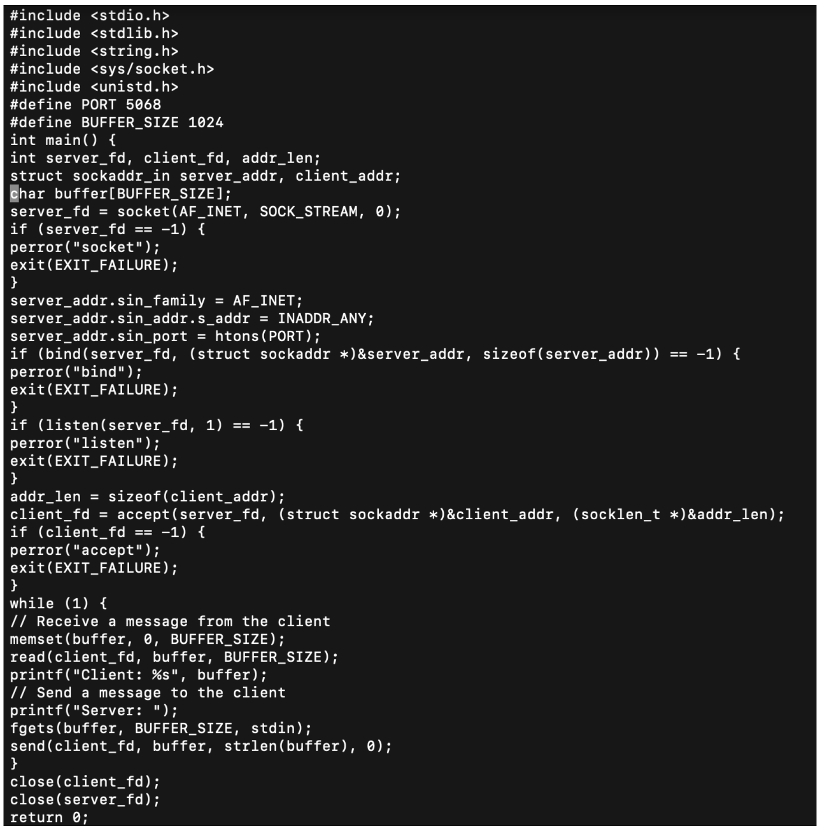
Question 1:
This is a simple chat programme that utilises server.c and client.c template, we simply have to modify the port and the address. However, in this case my VM has broken down, therefore I have opted to use a local machine that uses loopback 127.0.0.1 and a designated port number to achieve the following outcome:



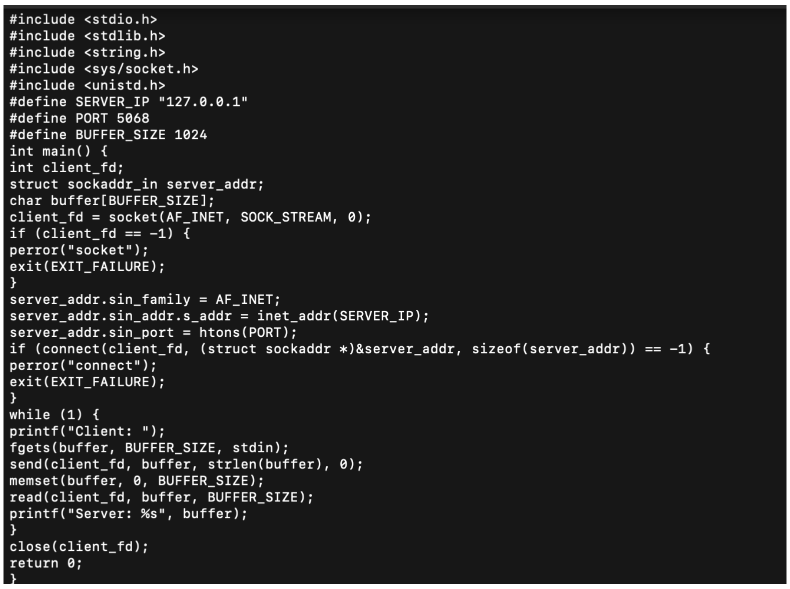
```
vellichastrxism@Vellichs-MacBook-Air Desktop % ./server
Client: Hello, This is a blocking message from the client
Server: Hi, I am responding to your message, although I ca
nnot type more than one message at once.
Client: That is okay!
Server: Hello!
Client: Hello!
Server: ^C
vellichastrxism@Vellichs-MacBook-Air Desktop % ./server
Client: Hello!
Server: Hi!
Can only respond to one messages at a time!
Client: Yes!
Server: 
```



```
vellichastrxism@dyn-118-138-108-8 Desktop % ./client
Client: Hello, This is a blocking message from the client
Client: That is okay!
Client: Hello!
Client: ^C
vellichastrxism@dyn-118-138-108-8 Desktop % ./client
Client: Hello!
Server: Hi!
Client: Yes!
Server: Can only respond to one messages at a time!
Client: 
```

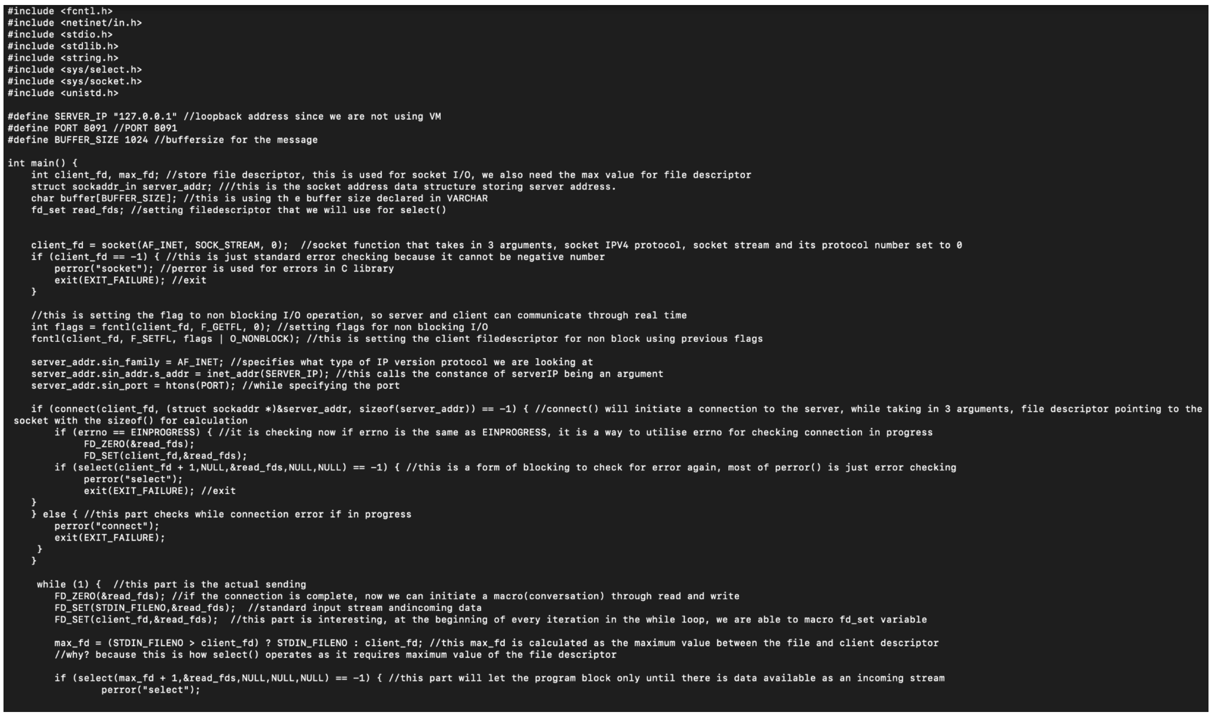


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#define PORT 5068
#define BUFFER_SIZE 1024
int main() {
int server_fd, client_fd, addr_len;
struct sockaddr_in server_addr, client_addr;
char buffer[BUFFER_SIZE];
server_fd = socket(AF_INET, SOCK_STREAM, 0);
if (server_fd == -1) {
perror("socket");
exit(EXIT_FAILURE);
}
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY;
server_addr.sin_port = htons(PORT);
if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
perror("bind");
exit(EXIT_FAILURE);
}
if (listen(server_fd, 1) == -1) {
perror("listen");
exit(EXIT_FAILURE);
}
addr_len = sizeof(client_addr);
client_fd = accept(server_fd, (struct sockaddr *)&client_addr, (socklen_t *)&addr_len);
if (client_fd == -1) {
perror("accept");
exit(EXIT_FAILURE);
}
while (1) {
// Receive a message from the client
memset(buffer, 0, BUFFER_SIZE);
read(client_fd, buffer, BUFFER_SIZE);
printf("Client: %s", buffer);
// Send a message to the client
printf("Server: ");
fgets(buffer, BUFFER_SIZE, stdin);
send(client_fd, buffer, strlen(buffer), 0);
}
close(client_fd);
close(server_fd);
return 0;
```

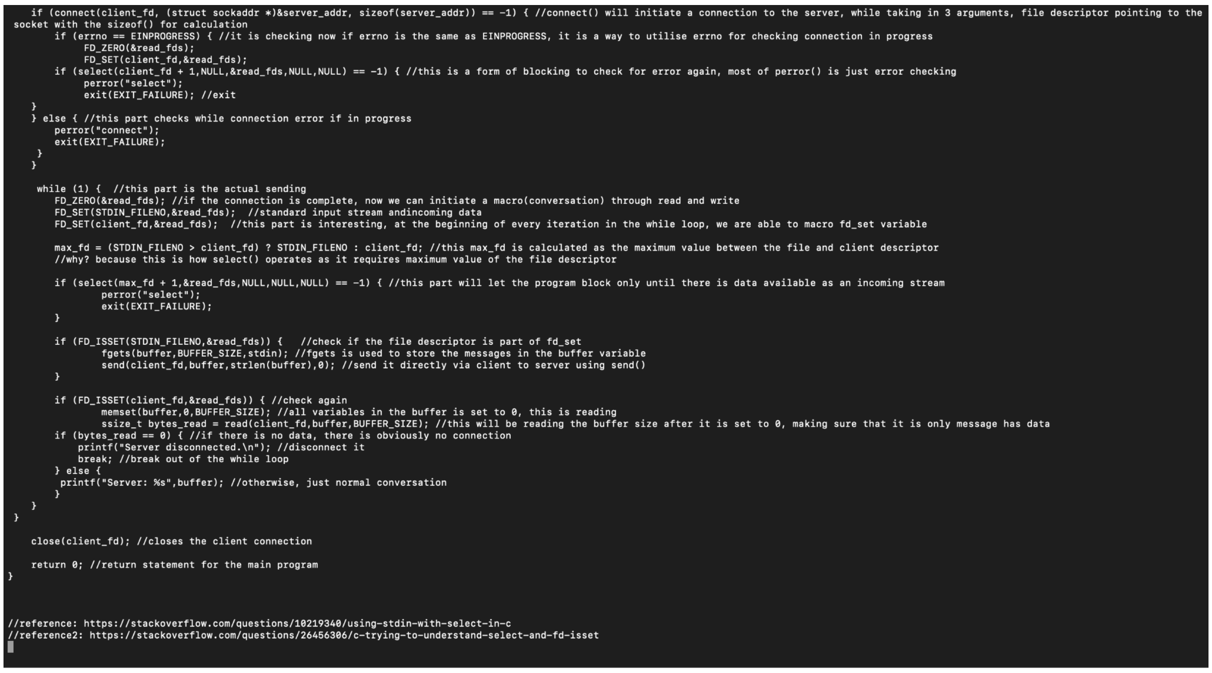


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>
#define SERVER_IP "127.0.0.1"
#define PORT 5068
#define BUFFER_SIZE 1024
int main() {
int client_fd;
struct sockaddr_in server_addr;
char buffer[BUFFER_SIZE];
client_fd = socket(AF_INET, SOCK_STREAM, 0);
if (client_fd == -1) {
perror("socket");
exit(EXIT_FAILURE);
}
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = inet_addr(SERVER_IP);
server_addr.sin_port = htons(PORT);
if (connect(client_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) {
perror("connect");
exit(EXIT_FAILURE);
}
while (1) {
printf("Client: ");
fgets(buffer, BUFFER_SIZE, stdin);
send(client_fd, buffer, strlen(buffer), 0);
memset(buffer, 0, BUFFER_SIZE);
read(client_fd, buffer, BUFFER_SIZE);
printf("Server: %s", buffer);
}
close(client_fd);
return 0;
}
```

Question 2:
newclient.c:



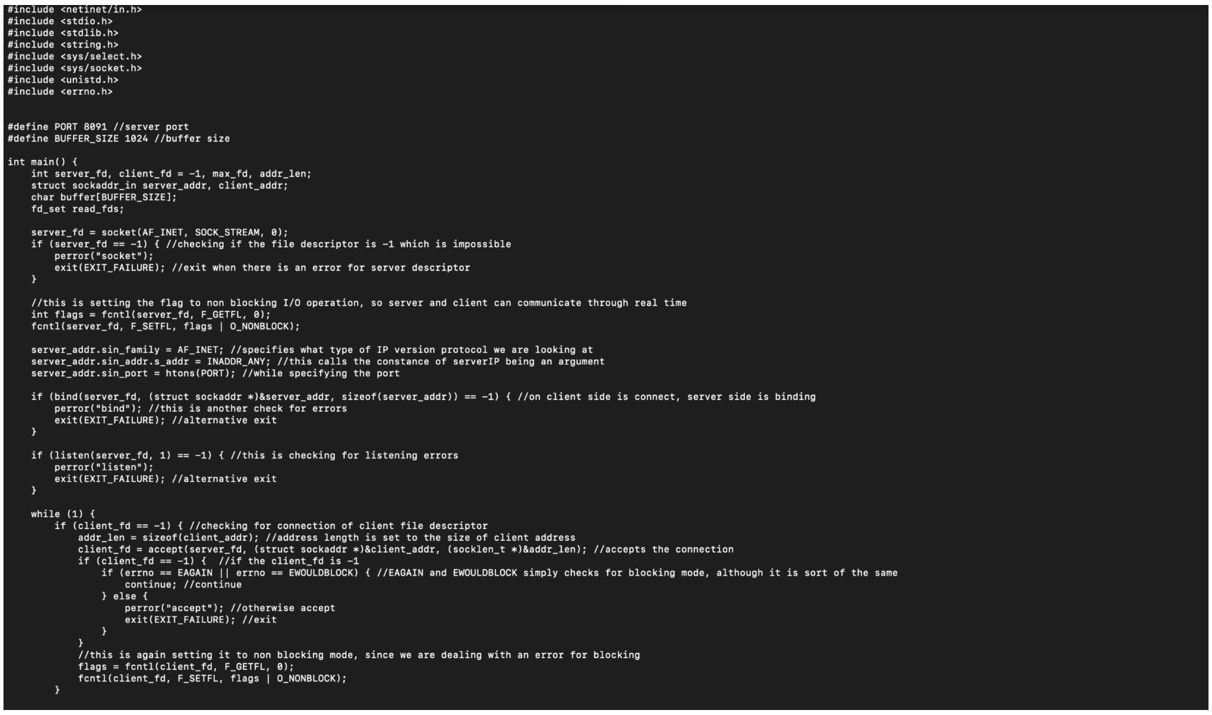
```c
#include <fcntl.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <unistd.h>

#define SERVER_IP "127.0.0.1" //loopback address since we are not using VM
#define PORT 8091 //PORT 8091
#define BUFFER_SIZE 1024 //buffersize for the message

int main() {
    int client_fd, max_fd; //store file descriptor, this is used for socket I/O, we also need the max value for file descriptor
    struct sockaddr_in server_addr; ///this is the socket address data structure storing server address.
    char buffer[BUFFER_SIZE]; //this is using th e buffer size declared in VARCHAR
    fd_set read_fds; //setting filedescriptor that we will use for select()


    client_fd = socket(AF_INET, SOCK_STREAM, 0);  //socket function that takes in 3 arguments, socket IPV4 protocol, socket stream and its protocol number set to 0
    if (client_fd == -1) { //this is just standard error checking because it cannot be negative number
        perror("socket"); //perror is used for errors in C library
        exit(EXIT_FAILURE); //exit
    }

    //this is setting the flag to non blocking I/O operation, so server and client can communicate through real time
    int flags = fcntl(client_fd, F_GETFL, 0); //setting flags for non blocking I/O
    fcntl(client_fd, F_SETFL, flags | O_NONBLOCK); //this is setting the client filedescriptor for non block using previous flags

    server_addr.sin_family = AF_INET; //specifies what type of IP version protocol we are looking at
    server_addr.sin_addr.s_addr = inet_addr(SERVER_IP); //this calls the constance of serverIP being an argument
    server_addr.sin_port = htons(PORT); //while specifying the port

    if (connect(client_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) { //connect() will initiate a connection to the server, while taking in 3 arguments, file descriptor pointing to the
 socket with the sizeof() for calculation
        if (errno == EINPROGRESS) { //it is checking now if errno is the same as EINPROGRESS, it is a way to utilise errno for checking connection in progress
             FD_ZERO(&read_fds);
             FD_SET(client_fd,&read_fds);
        if (select(client_fd + 1,NULL,&read_fds,NULL,NULL) == -1) { //this is a form of blocking to check for error again, most of perror() is just error checking
             perror("select");
             exit(EXIT_FAILURE); //exit
    }
    } else { //this part checks while connection error if in progress
        perror("connect");
        exit(EXIT_FAILURE);
     }
    }

     while (1) {  //this part is the actual sending
        FD_ZERO(&read_fds); //if the connection is complete, now we can initiate a macro(conversation) through read and write
        FD_SET(STDIN_FILENO,&read_fds);  //standard input stream andincoming data
        FD_SET(client_fd,&read_fds);  //this part is interesting, at the beginning of every iteration in the while loop, we are able to macro fd_set variable

        max_fd = (STDIN_FILENO > client_fd) ? STDIN_FILENO : client_fd; //this max_fd is calculated as the maximum value between the file and client descriptor
        //why? because this is how select() operates as it requires maximum value of the file descriptor

        if (select(max_fd + 1,&read_fds,NULL,NULL,NULL) == -1) { //this part will let the program block only until there is data available as an incoming stream
                perror("select");
```



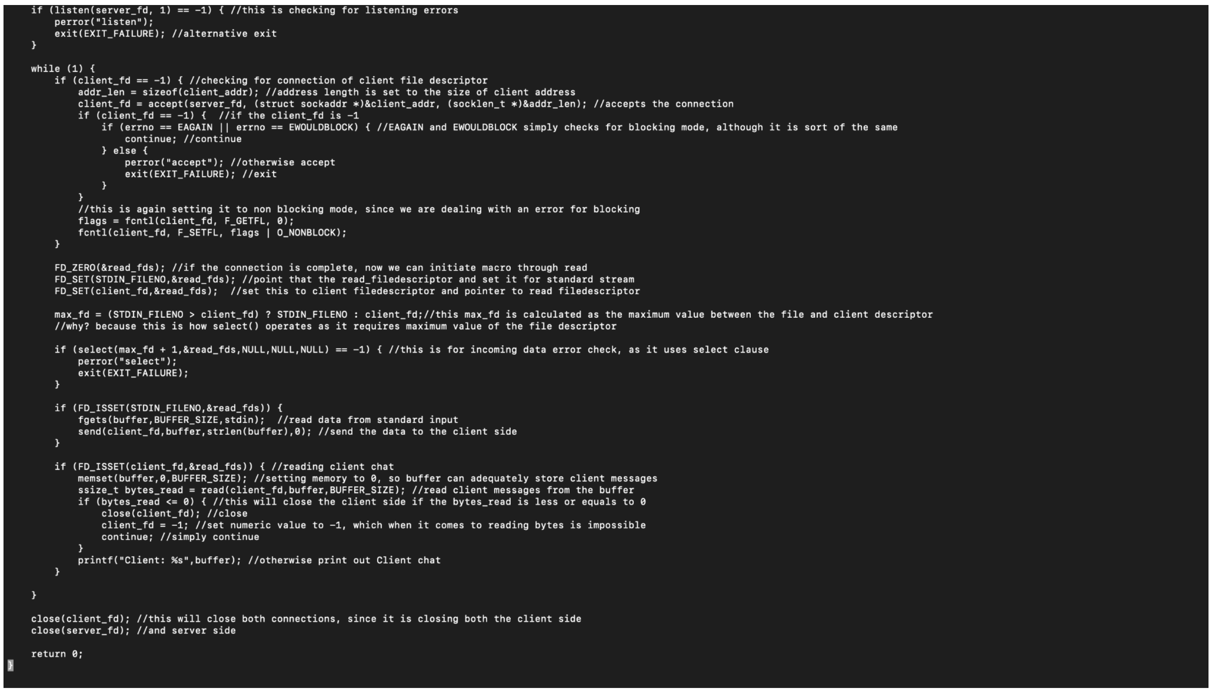
```c
    if (connect(client_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) { //connect() will initiate a connection to the server, while taking in 3 arguments, file descriptor pointing to the
 socket with the sizeof() for calculation
        if (errno == EINPROGRESS) { //it is checking now if errno is the same as EINPROGRESS, it is a way to utilise errno for checking connection in progress
             FD_ZERO(&read_fds);
             FD_SET(client_fd,&read_fds);
        if (select(client_fd + 1,NULL,&read_fds,NULL,NULL) == -1) { //this is a form of blocking to check for error again, most of perror() is just error checking
             perror("select");
             exit(EXIT_FAILURE); //exit
    }
    } else { //this part checks while connection error if in progress
        perror("connect");
        exit(EXIT_FAILURE);
     }
    }

     while (1) {  //this part is the actual sending
        FD_ZERO(&read_fds); //if the connection is complete, now we can initiate a macro(conversation) through read and write
        FD_SET(STDIN_FILENO,&read_fds);  //standard input stream andincoming data
        FD_SET(client_fd,&read_fds);  //this part is interesting, at the beginning of every iteration in the while loop, we are able to macro fd_set variable

        max_fd = (STDIN_FILENO > client_fd) ? STDIN_FILENO : client_fd; //this max_fd is calculated as the maximum value between the file and client descriptor
        //why? because this is how select() operates as it requires maximum value of the file descriptor

        if (select(max_fd + 1,&read_fds,NULL,NULL,NULL) == -1) { //this part will let the program block only until there is data available as an incoming stream
                perror("select");
```

```c
                exit(EXIT_FAILURE);
        }

        if (FD_ISSET(STDIN_FILENO,&read_fds)) {   //check if the file descriptor is part of fd_set
                fgets(buffer,BUFFER_SIZE,stdin); //fgets is used to store the messages in the buffer variable
                send(client_fd,buffer,strlen(buffer),0); //send it directly via client to server using send()
        }

        if (FD_ISSET(client_fd,&read_fds)) { //check again
                memset(buffer,0,BUFFER_SIZE); //all variables in the buffer is set to 0, this is reading
                ssize_t bytes_read = read(client_fd,buffer,BUFFER_SIZE); //this will be reading the buffer size after it is set to 0, making sure that it is only message has data
        if (bytes_read == 0) { //if there is no data, there is obviously no connection
            printf("Server disconnected.\n"); //disconnect it
            break; //break out of the while loop
        } else {
         printf("Server: %s",buffer); //otherwise, just normal conversation
        }
    }
 }

    close(client_fd); //closes the client connection

    return 0; //return statement for the main program
}


//reference: https://stackoverflow.com/questions/10219340/using-stdin-with-select-in-c
//reference2: https://stackoverflow.com/questions/26456306/c-trying-to-understand-select-and-fd-isset
```

newserver.c



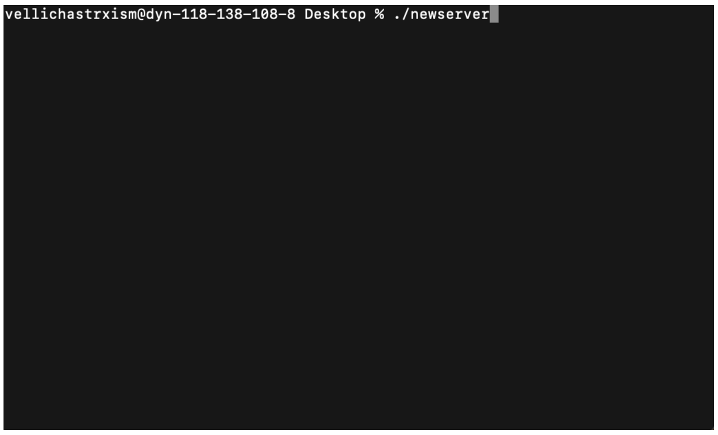
```c
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <unistd.h>
#include <errno.h>


#define PORT 8091 //server port
#define BUFFER_SIZE 1024 //buffer size

int main() {
    int server_fd, client_fd = -1, max_fd, addr_len;
    struct sockaddr_in server_addr, client_addr;
    char buffer[BUFFER_SIZE];
    fd_set read_fds;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) { //checking if the file descriptor is -1 which is impossible
        perror("socket");
        exit(EXIT_FAILURE); //exit when there is an error for server descriptor
    }

    //this is setting the flag to non blocking I/O operation, so server and client can communicate through real time
    int flags = fcntl(server_fd, F_GETFL, 0);
    fcntl(server_fd, F_SETFL, flags | O_NONBLOCK);

    server_addr.sin_family = AF_INET; //specifies what type of IP version protocol we are looking at
    server_addr.sin_addr.s_addr = INADDR_ANY; //this calls the constance of serverIP being an argument
    server_addr.sin_port = htons(PORT); //while specifying the port

    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) == -1) { //on client side is connect, server side is binding
        perror("bind"); //this is another check for errors
        exit(EXIT_FAILURE); //alternative exit
    }

    if (listen(server_fd, 1) == -1) { //this is checking for listening errors
        perror("listen");
        exit(EXIT_FAILURE); //alternative exit
    }

    while (1) {
        if (client_fd == -1) { //checking for connection of client file descriptor
            addr_len = sizeof(client_addr); //address length is set to the size of client address
            client_fd = accept(server_fd, (struct sockaddr *)&client_addr, (socklen_t *)&addr_len); //accepts the connection
            if (client_fd == -1) {  //if the client_fd is -1
                if (errno == EAGAIN || errno == EWOULDBLOCK) { //EAGAIN and EWOULDBLOCK simply checks for blocking mode, although it is sort of the same
                    continue; //continue
                } else {
                    perror("accept"); //otherwise accept
                    exit(EXIT_FAILURE); //exit
                }
            }
            //this is again setting it to non blocking mode, since we are dealing with an error for blocking
            flags = fcntl(client_fd, F_GETFL, 0);
            fcntl(client_fd, F_SETFL, flags | O_NONBLOCK);
        }
```



```c
    if (listen(server_fd, 1) == -1) { //this is checking for listening errors
        perror("listen");
        exit(EXIT_FAILURE); //alternative exit
    }

    while (1) {
        if (client_fd == -1) { //checking for connection of client file descriptor
            addr_len = sizeof(client_addr); //address length is set to the size of client address
            client_fd = accept(server_fd, (struct sockaddr *)&client_addr, (socklen_t *)&addr_len); //accepts the connection
            if (client_fd == -1) {  //if the client_fd is -1
                if (errno == EAGAIN || errno == EWOULDBLOCK) { //EAGAIN and EWOULDBLOCK simply checks for blocking mode, although it is sort of the same
                    continue; //continue
                } else {
                    perror("accept"); //otherwise accept
                    exit(EXIT_FAILURE); //exit
                }
            }
            //this is again setting it to non blocking mode, since we are dealing with an error for blocking
            flags = fcntl(client_fd, F_GETFL, 0);
            fcntl(client_fd, F_SETFL, flags | O_NONBLOCK);
        }
```

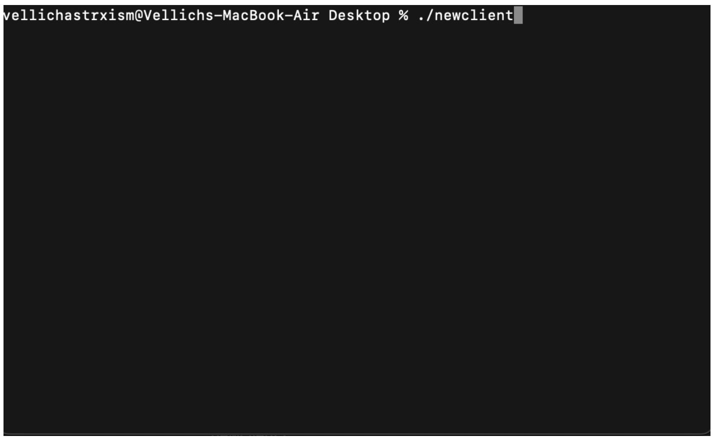
```c

        FD_ZERO(&read_fds); //if the connection is complete, now we can initiate macro through read
        FD_SET(STDIN_FILENO,&read_fds); //point that the read_filedescriptor and set it for standard stream
        FD_SET(client_fd,&read_fds);  //set this to client filedescriptor and pointer to read filedescriptor

        max_fd = (STDIN_FILENO > client_fd) ? STDIN_FILENO : client_fd;//this max_fd is calculated as the maximum value between the file and client descriptor
        //why? because this is how select() operates as it requires maximum value of the file descriptor

        if (select(max_fd + 1,&read_fds,NULL,NULL,NULL) == -1) { //this is for incoming data error check, as it uses select clause
            perror("select");
            exit(EXIT_FAILURE);
        }

        if (FD_ISSET(STDIN_FILENO,&read_fds)) {
            fgets(buffer,BUFFER_SIZE,stdin);  //read data from standard input
            send(client_fd,buffer,strlen(buffer),0); //send the data to the client side
        }

        if (FD_ISSET(client_fd,&read_fds)) { //reading client chat
            memset(buffer,0,BUFFER_SIZE); //setting memory to 0, so buffer can adequately store client messages
            ssize_t bytes_read = read(client_fd,buffer,BUFFER_SIZE); //read client messages from the buffer
            if (bytes_read <= 0) { //this will close the client side if the bytes_read is less or equals to 0
                close(client_fd); //close
                client_fd = -1; //set numeric value to -1, which when it comes to reading bytes is impossible
                continue; //simply continue
            }
            printf("Client: %s",buffer); //otherwise print out Client chat
        }

    }

    close(client_fd); //this will close both connections, since it is closing both the client side
    close(server_fd); //and server side

    return 0;
}
```

Question 3:
Compiling newserver.c, where newserver.c will be the server side

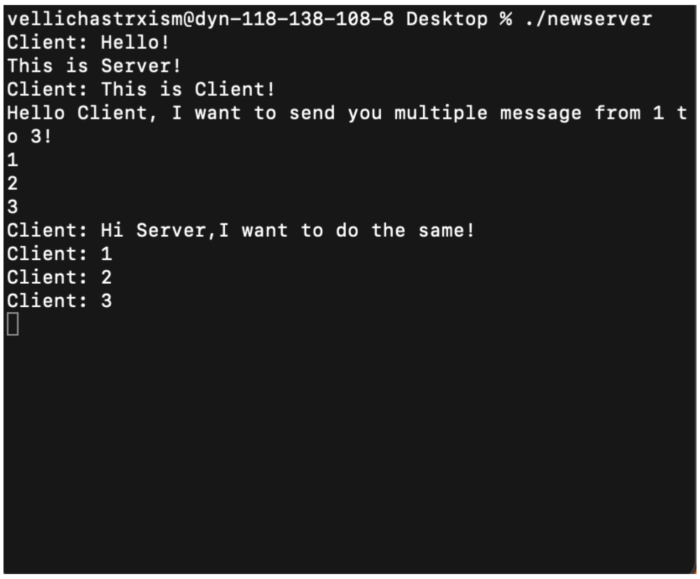


```
vellichastrxism@dyn-118-138-108-8 Desktop % ./newserver
```

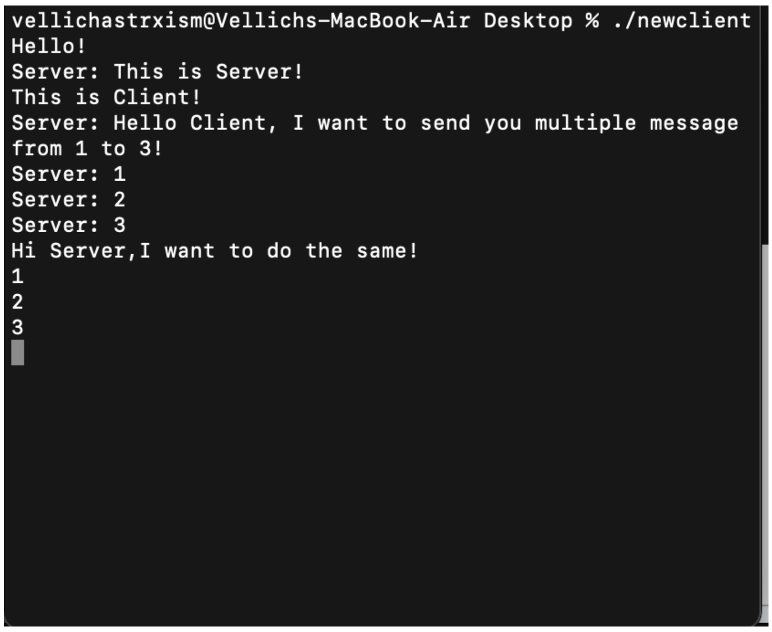
Compiling newclient.c where newclient will be the client side:



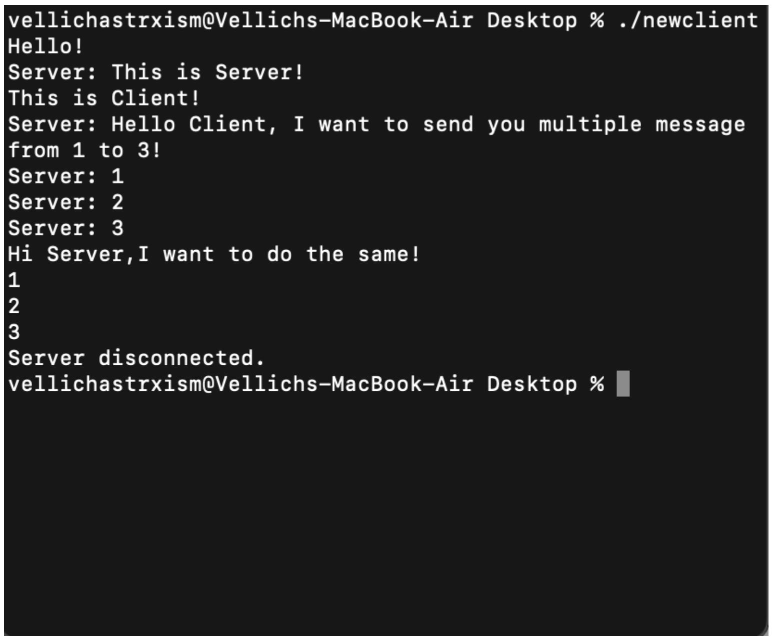
```
vellichastrxism@Vellichs-MacBook-Air Desktop % ./newclient
```



```
vellichastrxism@dyn-118-138-108-8 Desktop % ./newserver
Client: Hello!
This is Server!
Client: This is Client!
Hello Client, I want to send you multiple message from 1 t
o 3!
1
2
3
Client: Hi Server,I want to do the same!
Client: 1
Client: 2
Client: 3
```



```
vellichastrxism@Vellichs-MacBook-Air Desktop % ./newclient
Hello!
Server: This is Server!
This is Client!
Server: Hello Client, I want to send you multiple message
from 1 to 3!
Server: 1
Server: 2
Server: 3
Hi Server,I want to do the same!
1
2
3
```



```
vellichastrxism@Vellichs-MacBook-Air Desktop % ./newclient
Hello!
Server: This is Server!
This is Client!
Server: Hello Client, I want to send you multiple message
from 1 to 3!
Server: 1
Server: 2
Server: 3
Hi Server,I want to do the same!
1
2
3
Server disconnected.
vellichastrxism@Vellichs-MacBook-Air Desktop %
```

The above two screenshots indicate a real time communication between client and server, as seen in the screenshot, both client and server can send multiple messages to each other through a common port connection with the respective server IP (loopback 127.0.0.1).

Once the server disconnects, on the client side it will display a message saying that the server has been disconnected.