ONLINE MULTIPLAYER SNAKES

CPS2003

APRIL 26, 2017

DYLAN GALEA
84296M

Table of Contents

Introduction	1
Design Description	2
Description of Protocols Used	5
Important Implementation Details	9
Serialization	*9*
Locks and Join	*9*
Implementation of the Snake Movement	*11*
Server Decisions	*12*
Test Cases & Results	13
Single Player Testing	*13*
Multiplayer Testing	*17*
List of Improvements & Bugs	25
Bugs	*25*
Improvements	*26*
Conclusion	27
References	28

Introduction

The aim of this study unit and assignment was to learn about different mechanisms to create solutions that are efficient and correct at the same time. In this assignment, an online Multiplayer snakes game had to implemented, thus although the correct execution of the program was a necessity, it was also important to implement mechanisms such as Mutex Locks and internet protocols for communication. Also, another important construct to be considered was multi-threading. According to [1], multithreading is the wave of the future since nowadays software solutions and companies require their software to be very efficient and have great performance since nowadays CPU's are increasing in multicores, thus the ability of performing many tasks simultaneously. This assignment was carried out in C for UNIX operating systems and per [2], C is a general-purpose language which is closely associated with the UNIX operating system, in fact many programs that run it are written in C. C was written in 1972 by Dennis Ritchie at Bell Labs and it was later published in 1978 by Kernighan & Ritchie. Later in 1983 ANSI updated the C library to provide a modern and more robust version of C and a result we call this the ANSI C. Thus, C is used heavily in systems programming.

As per [3], Systems programming deals with concepts that are more related to the hardware level such as memory and is used to control and manage a computer system. System's programming also deals with many concepts for communication to provide efficiency and control between different processes.

This write up is divided into 5 sections, the first section is about the design concepts of the solution, the second section is about the implemented protocols, third section describes some important mechanisms in the solution , the test cases considered and test results and a list of bugs and improvements that could be made. Thus, in this write up there is not written any specific implementation details, but to read about them see the code listings along with the comments that describe each execution flow. The following section will describe the system design and how different components interact with each other.

Design Description

The multiplayer snakes game was divided into 4 major components as can be seen from the code listings. Namely these was the game file, connection file, server file and client file. The game file contains all the game related information and methods to be performed by the server to incorporate the game logic together, combined with the related connection details and mutex locks mechanisms. The game file has several constants that were declared so that whenever a change was needed or is needed in the game the code does not need modification, such constants are the controls, snake body character, food body character etc. Four structures that were declared in the game file were the 'element','snakeobject','player' and 'game' struct. The element struct simply contains the co-ordinates on the ncurses terminal that need to be displayed with a special character, the snake object struct contains the snake details which are the snake size, delay the snake has for the game to speed up if the snake ate enough food, ate food variable that indicates that the snake needs to grow since it ate food and the direction it is going. This structure also contains the snake body which is an array of coordinates that make up the snake. To see how the snake is moving see the 3rd section in this report below. Another structure in the game file was the player structure which contains player information such as the number of points he has and whether he's killed or not. The killed variable is important in the server because once a player is killed he must be ignored for collisions and in the client, he cannot be displayed since he is no longer part of the game, also if the client is killed he must terminate the connection. The last structure in the game file is the game struct which contains the relative game information such as the players that are currently playing, were the food is residing in the game etc. It is important to note that once a client is killed he is not removed from the list of players since we have an array and this would create many movements in memory to keep it clean, thus when a player is killed he is made to be ignored in the server. The game file also has certain methods so that the server can create a snake, determine the starting position, check for collisions, moving the snake etc.

The next component of this multiplayer online snakes' game is the connection file, where this file contains all the necessary methods and structs to create a connection between the server and multiple clients. First 2 structures for the client and server separately were created where these contain the necessary variable connection information for the server and client respectively. This was done so that no variables are running around in server and client. Also, methods were created for both server and client to create a TCP/IP socket, binding, connecting to the server and closing the server and client end of communication. Another thing worth mentioning, also described in section 3, was that to send information over TCP/IP structs could not be sent hence they had to be deserialized and the receiving end and serialized in the sending end for correct communication, this was done as the server always sent the game struct to the clients.

The server was basically designed to first create a TCP/IP socket, then bind it and blocks to listen for the first connection. This was done because the game cannot be started unless a first connection is given, then since the server was created to be multi-threaded an extra thread was created to always block and listen for more connections whilst at the same time the server can move other snakes and check for collisions in the game. Thus, after creating the first connection the server creates multiple threads to get the user input and waits for the threads to finish to not create conflicts. This is important because if in the main thread the server checks for collisions before a snake moves, then that snake can't avoid being killed because the user is not given chance to move. Thus prior to check for collisions all threads must be joined. Then the server checks for collisions and then creates more threads to move the snakes to perform multiple requests at the same time, and for the same reason above waits for the threads to finish execution. It is also important to note that before creating the threads for movement, reading and waiting for connection a lock is created so that whilst a new snake is being created and is still not initialized the server does not move an undefined snake since this could break the game. It is also important to note that after each update to the snakes the server sends the whole game structure to the respective clients. Thus, therefore serialization was important. It is also important to note that the server keeps on restarting the game until all clients that have been connected have lost in which case the server stops execution, this was done so to keep everything simple and to make sure that the server closes cleanly.

The last component of this game is the client file, the client is a simple program that sends the client input from the ncurses terminal to the server, it is also important to note that the client was done to be multithreaded, in fact the terminal input is read in a separate thread. After sending the client input the client program blocks until the server sends the game struct and hence then updates his screen.

In figure 1- below the basic game structure could be seen, where the arrows below are connected via a TCP/IP connection. The bullet points in each entity show the different thread of executions that do different requests as required. It is important to note again that the 4 bullets in the server part must be done separately to avoid collisions and race conditions, thus as described mutex locks and waiting for threads to finish were used.

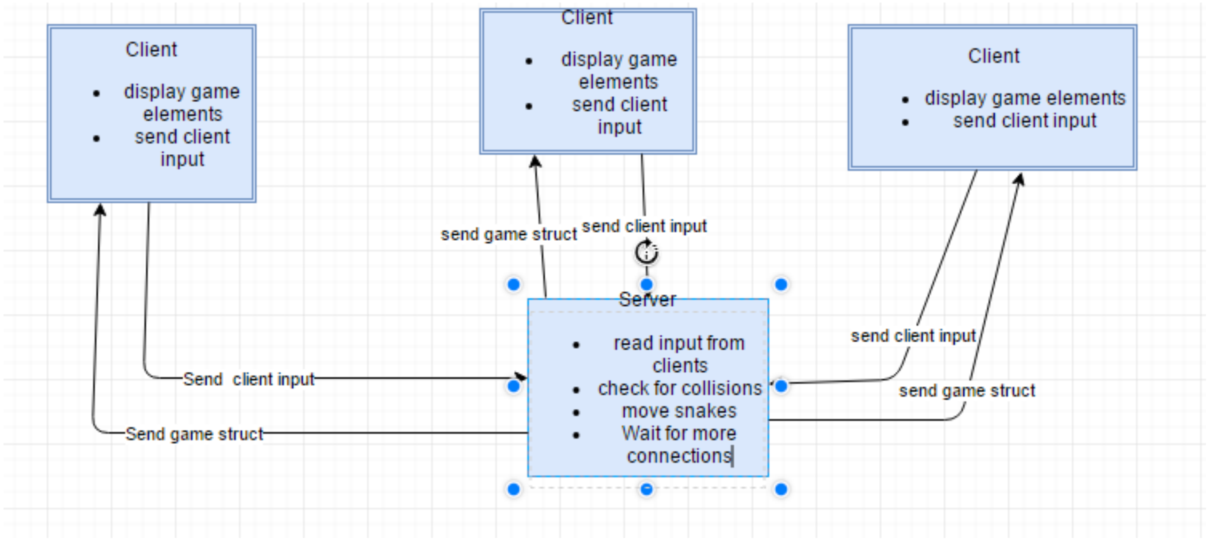
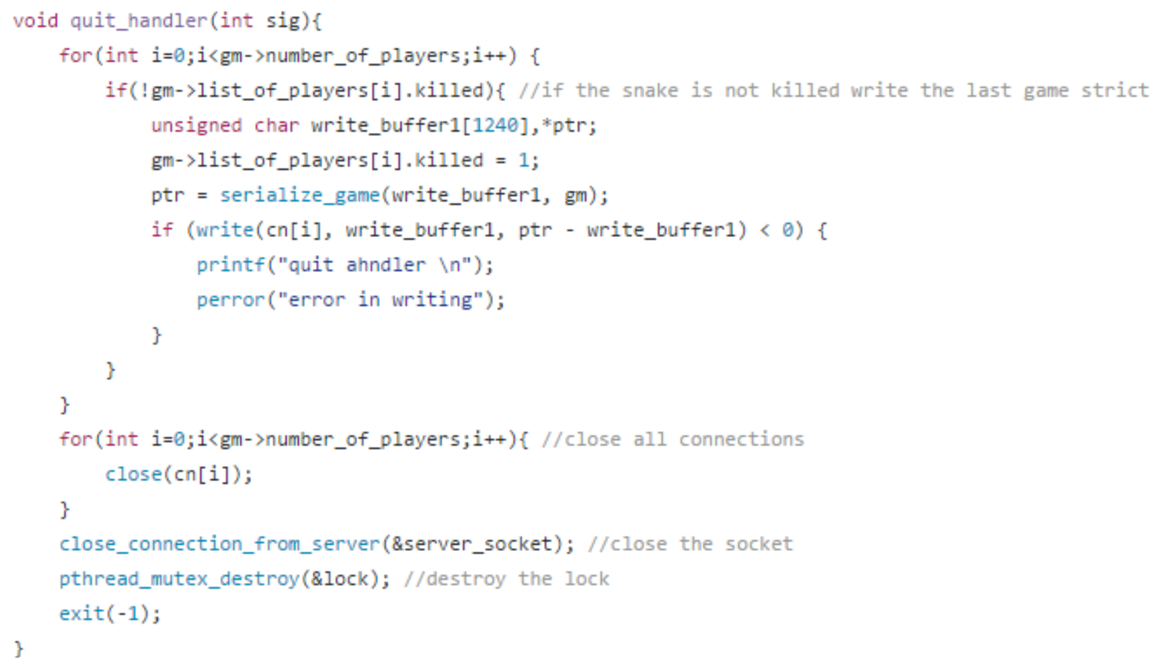


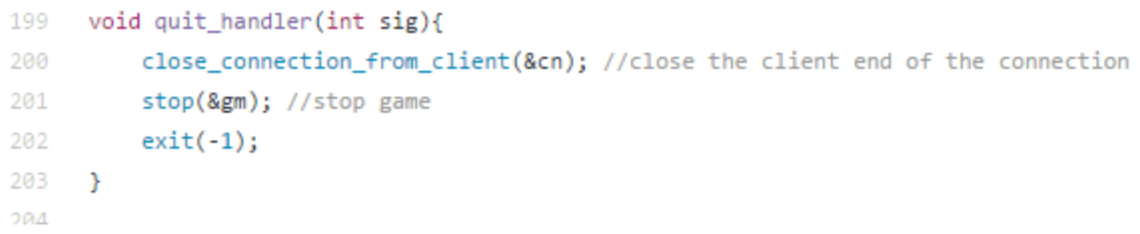
Figure 1 Overall Basic design of the system

Another design concept is being that termination had to be handled correctly along with signals that could be received. A quit-handler as shown in figure 2 below was designed so that whenever a SIGINT signal or any other signal is received the server can terminate cleanly by closing all connections and stop the game. Also, same for the client as seen in figure 3 below whenever a client is given a signal he must stop without interfering with other clients and the server. It is important to note that as per [4], killing signals are not handled cleanly a thus although done for completion because it was required in the specification the killing of the server and client does not make them close cleanly.



```
void quit_handler(int sig){
    for(int i=0;i<gm->number_of_players;i++) {
        if(!gm->list_of_players[i].killed){ //if the snake is not killed write the last game strict
            unsigned char write_buffer1[1240],*ptr;
            gm->list_of_players[i].killed = 1;
            ptr = serialize_game(write_buffer1, gm);
            if (write(cn[i], write_buffer1, ptr - write_buffer1) < 0) {
                printf("quit ahndler \n");
                perror("error in writing");
            }
        }
    }
    for(int i=0;i<gm->number_of_players;i++){ //close all connections
        close(cn[i]);
    }
    close_connection_from_server(&server_socket); //close the socket
    pthread_mutex_destroy(&lock); //destroy the lock
    exit(-1);
}
```

Figure 2-Server Handler

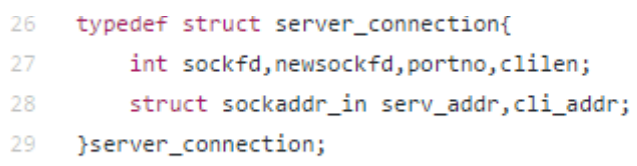


```
void quit_handler(int sig){
    close_connection_from_client(&cn); //close the client end of the connection
    stop(&gm); //stop game
    exit(-1);
}
```

Figure 3-Client Handler

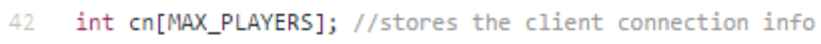
Description of Protocols Used

The only protocol used in this game is for the network protocol, namely the TCP/IP protocol, as per [5], the TCP/IP protocol provides an end to end connectivity specifying how data should be formatted, addressed, transmitted and how it should be received at the destination. Thus, in our design, the server and the client are bounded by this TCP/IP protocol, were the functions that implement this protocol could be found in the connection file. One, must also mention that in this implementation a newsockfd variable is used to contain the file descriptor of the last connection which has connected with the server, and each value is then put into an array in the server so that the server keeps track of the different connections to read and write to them. This detail can be seen in the figures below.



```
typedef struct server_connection{
    int sockfd,newsockfd,portno,clilen;
    struct sockaddr_in serv_addr,cli_addr;
}server_connection;
```

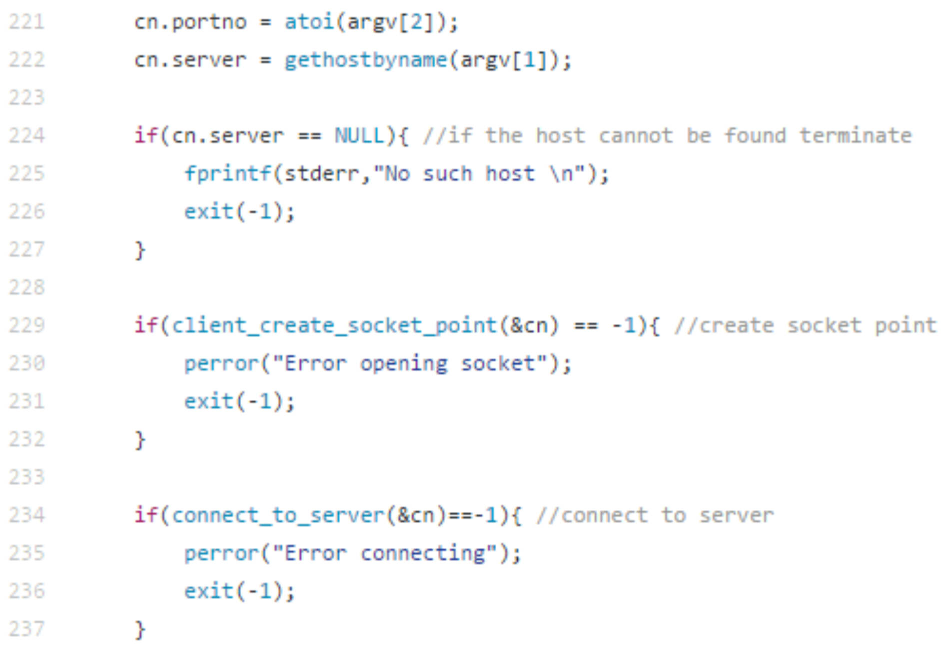
Figure 4-server structure - has newsockfd



```
int cn[MAX_PLAYERS]; //stores the client connection info
```

Figure 5-Array that contains all newsockfd's that connected with the client

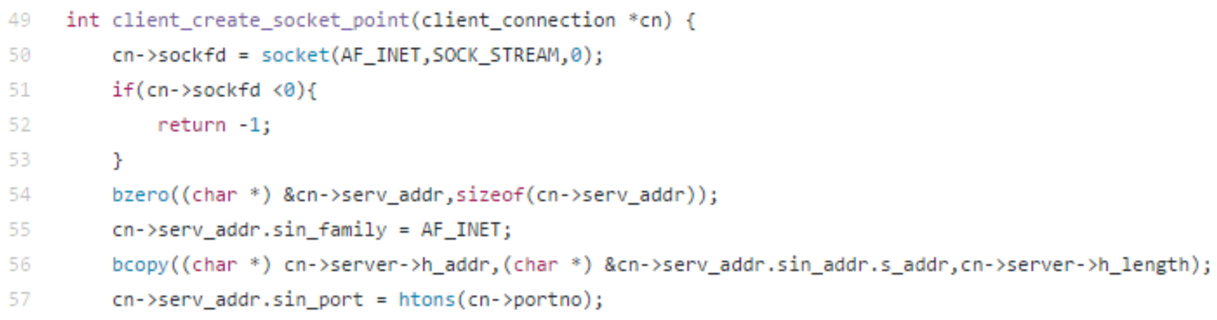
As per [6], the TCP/IP protocol is obviously based on the IP protocol where unlike in a single network were every computer is connected to each other one we have an inter-network i.e. a collection of more than 1 network which are connected to form part of a larger network. Thus, any computer on this large virtual network can communicate with other computers on the same virtual network through an address called the IP address. The IP address in fact is used in the provided implementation by the client to connect with the server over the internet as seen especially in lines 234-236 and 224-226 below:



```
    cn.portno = atoi(argv[2]);
    cn.server = gethostbyname(argv[1]);

    if(cn.server == NULL){ //if the host cannot be found terminate
        fprintf(stderr,"No such host \n");
        exit(-1);
    }

    if(client_create_socket_point(&cn) == -1){ //create socket point
        perror("Error opening socket");
        exit(-1);
    }

    if(connect_to_server(&cn)==-1){ //connect to server
        perror("Error connecting");
        exit(-1);
    }
```

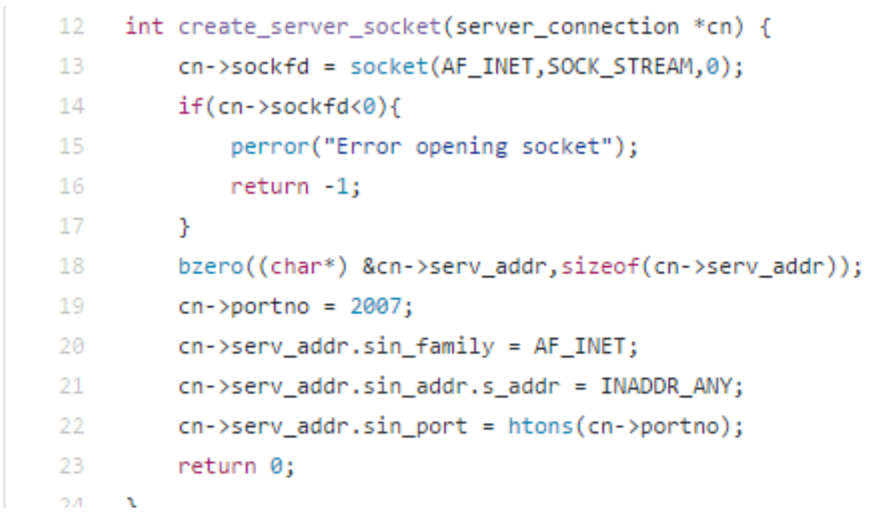
Figure 6

However, the IP protocol alone does not ensure that data is received correctly at the destination thus along with the IP protocol, the TCP protocol also ensures that the arriving packets are received at the same sequence they are sent and all duplicated packets are ignored. TCP also ensures that in the receiving end the sent packets are joined together into a stream of bytes. In this implementation, this protocol is implemented in the following figures below were byte ordering is done in lines 57 and 22 respectively in figures 7 and 8:



```
49  int client_create_socket_point(client_connection *cn) {
50      cn->sockfd = socket(AF_INET,SOCK_STREAM,0);
51      if(cn->sockfd <0){
52          return -1;
53      }
54      bzero((char *) &cn->serv_addr,sizeof(cn->serv_addr));
55      cn->serv_addr.sin_family = AF_INET;
56      bcopy((char *) cn->server->h_addr,(char *) &cn->serv_addr.sin_addr.s_addr,cn->server->h_length);
57      cn->serv_addr.sin_port = htons(cn->portno);
```

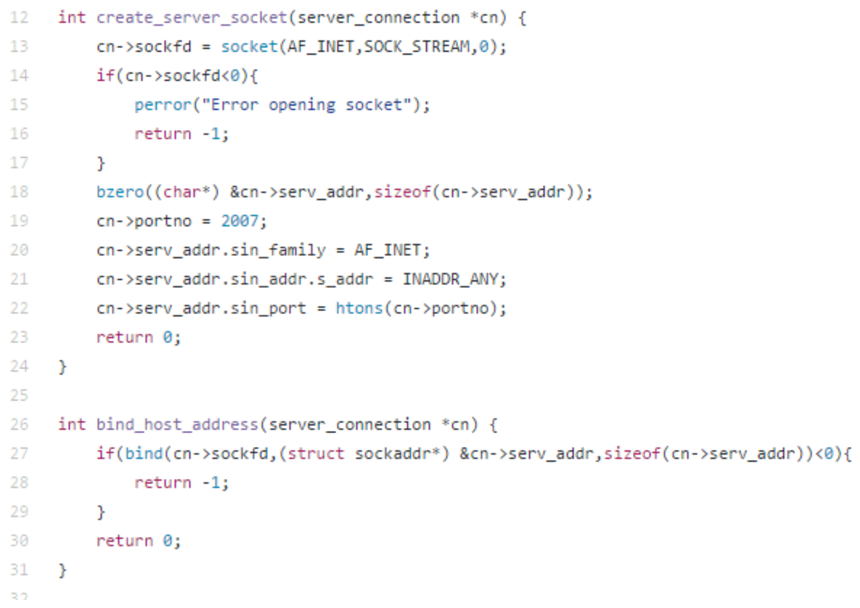
Figure 7



```c
int create_server_socket(server_connection *cn) {
    cn->sockfd = socket(AF_INET,SOCK_STREAM,0);
    if(cn->sockfd<0){
        perror("Error opening socket");
        return -1;
    }
    bzero((char*) &cn->serv_addr,sizeof(cn->serv_addr));
    cn->portno = 2007;
    cn->serv_addr.sin_family = AF_INET;
    cn->serv_addr.sin_addr.s_addr = INADDR_ANY;
    cn->serv_addr.sin_port = htons(cn->portno);
    return 0;
```

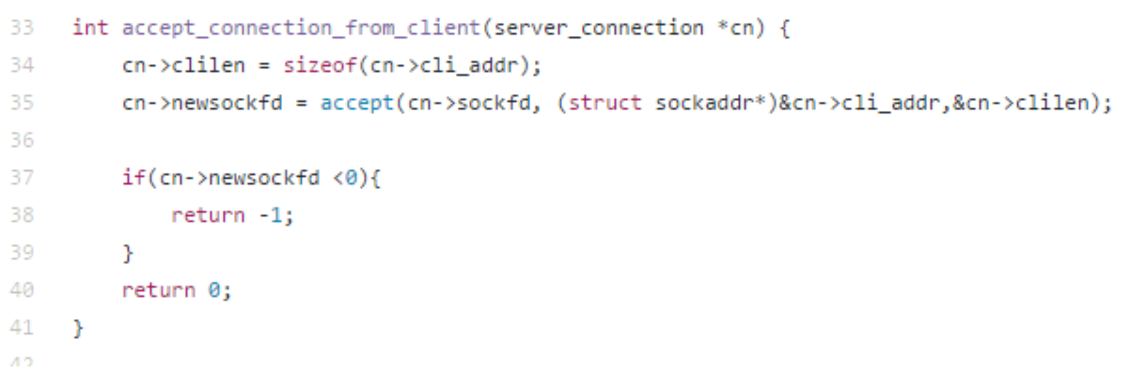
Figure 8

TCP is also a connection oriented protocol; hence a connection must be established first for the client and the server to communicate and one the connection is closed they are no longer connected. The server must first create a socket where a socket is a device used to communicate with the network. Thus, when bytes are to be written to the network they are written to the socket, and when bytes are to be read from the network, the socket must be read. Then the server must bind that socket to utilize a specific port or address. The server then must listen for connections from the clients. Thus, in this implementation of the game another thread is created to listen for connections. Then the client's job is to create a connection with the server using the IP of the server and the port it wants to connect to, and then server the accepts it. Afterwards data can be sent and be received from both the server and the client till the connection stops. In the figure below the functions implemented in the connection file could be seen which complement the description in this paragraph.



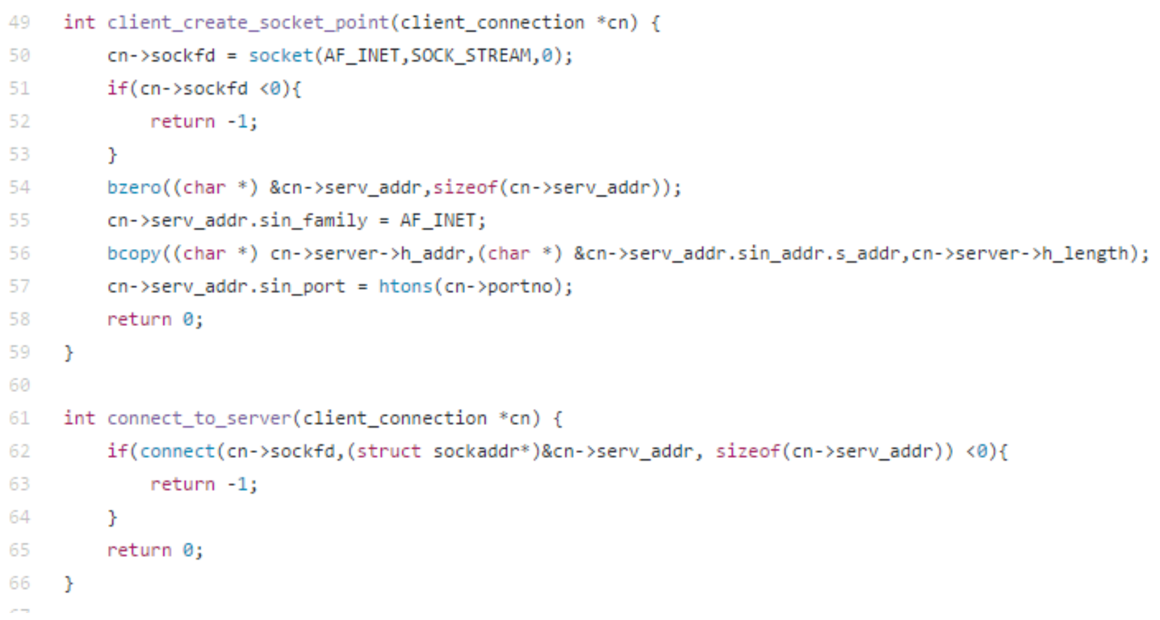
```c
12  int create_server_socket(server_connection *cn) {
13      cn->sockfd = socket(AF_INET,SOCK_STREAM,0);
14      if(cn->sockfd<0){
15          perror("Error opening socket");
16          return -1;
17      }
18      bzero((char*) &cn->serv_addr,sizeof(cn->serv_addr));
19      cn->portno = 2007;
20      cn->serv_addr.sin_family = AF_INET;
21      cn->serv_addr.sin_addr.s_addr = INADDR_ANY;
22      cn->serv_addr.sin_port = htons(cn->portno);
23      return 0;
24  }
25
26  int bind_host_address(server_connection *cn) {
27      if(bind(cn->sockfd,(struct sockaddr*) &cn->serv_addr,sizeof(cn->serv_addr))<0){
28          return -1;
29      }
30      return 0;
31  }
```

Figure 9-lines 12-24 creation of the socket from the server, 26-31 binding of the socket



```c
int accept_connection_from_client(server_connection *cn) {
    cn->clilen = sizeof(cn->cli_addr);
    cn->newsockfd = accept(cn->sockfd, (struct sockaddr*)&cn->cli_addr,&cn->clilen);

    if(cn->newsockfd <0){
        return -1;
    }
    return 0;
}
```

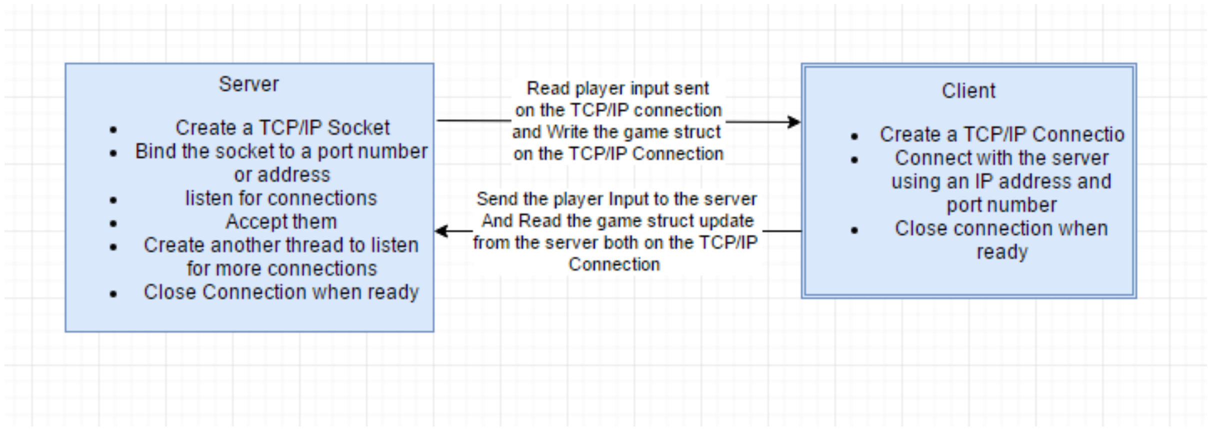
Figure 10-by this method the server blocks until a connection is sent by the server in which case it is then accepted



```c
49  int client_create_socket_point(client_connection *cn) {
50      cn->sockfd = socket(AF_INET,SOCK_STREAM,0);
51      if(cn->sockfd <0){
52          return -1;
53      }
54      bzero((char *) &cn->serv_addr,sizeof(cn->serv_addr));
55      cn->serv_addr.sin_family = AF_INET;
56      bcopy((char *) cn->server->h_addr,(char *) &cn->serv_addr.sin_addr.s_addr,cn->server->h_length);
57      cn->serv_addr.sin_port = htons(cn->portno);
58      return 0;
59  }
60
61  int connect_to_server(client_connection *cn) {
62      if(connect(cn->sockfd,(struct sockaddr*)&cn->serv_addr, sizeof(cn->serv_addr)) <0){
63          return -1;
64      }
65      return 0;
66  }
```

Figure 11-lines 49-59 connection is created and the in lines 61-66 the client tries to connect with the server

For an easier understanding of the TCP/IP Protocol in this implementation see the diagram below:



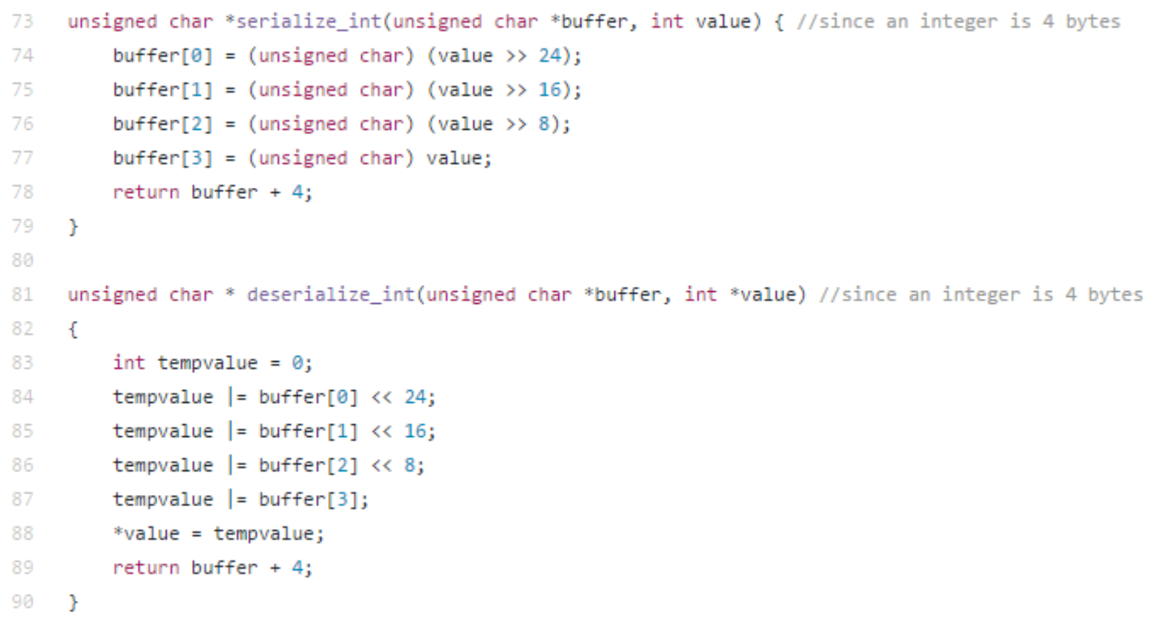
Server
Create a TCP/IP Socket
Bind the socket to a port number or address
listen for connections
Accept them
Create another thread to listen for more connections
Close Connection when ready
Read player input sent on the TCP/IP connection and Write the game struct on the TCP/IP Connection
Send the player Input to the server And Read the game struct update from the server both on the TCP/IP Connection
Client
Create a TCP/IP Connectio
Connect with the server using an IP address and port number
Close connection when ready

Figure 12-Describes briefly the TCP/IP connection between the server and the client .. the arrows represent the TCP/IP Connection when data flow from the sockets

Important Implementation Details

Serialization

One important mechanism used in this implementation is serialization. Serialization was used because the game map was designed to be contained in a struct and thus since structs could not be sent over a connection they had to first be converted to bytes in the sender's end and then converted back to the original struct in the receiver end. These serialization and deserialization mechanisms were implemented in the connection files and to see more specific details see the Dioxygen comments. In figure 13 below the serialization and deserialization routines are shown as an example:



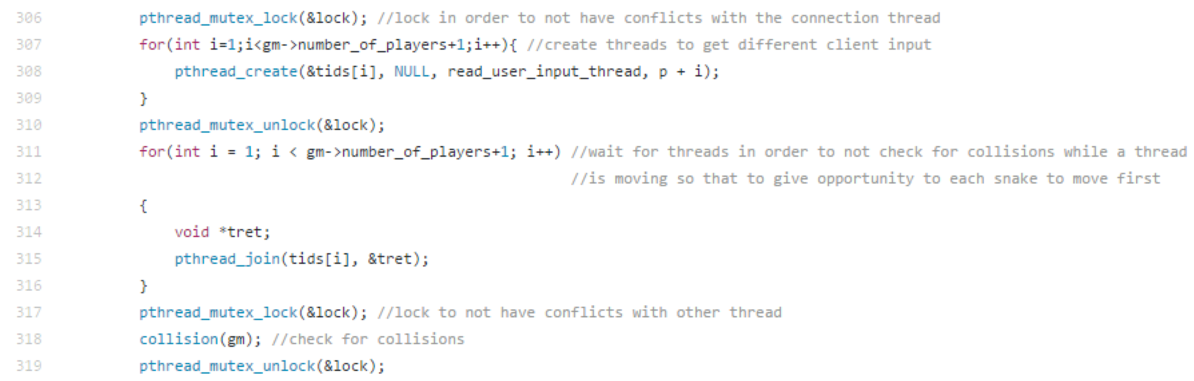
```
73  unsigned char *serialize_int(unsigned char *buffer, int value) { //since an integer is 4 bytes
74      buffer[0] = (unsigned char) (value >> 24);
75      buffer[1] = (unsigned char) (value >> 16);
76      buffer[2] = (unsigned char) (value >> 8);
77      buffer[3] = (unsigned char) value;
78      return buffer + 4;
79  }
80
81  unsigned char * deserialize_int(unsigned char *buffer, int *value) //since an integer is 4 bytes
82  {
83      int tempvalue = 0;
84      tempvalue |= buffer[0] << 24;
85      tempvalue |= buffer[1] << 16;
86      tempvalue |= buffer[2] << 8;
87      tempvalue |= buffer[3];
88      *value = tempvalue;
89      return buffer + 4;
90  }
```

Figure 13-Serialization and Deserialization of integer

As it can be seen in figure 13 above lines 74-78 first the variable 'value' containing the integer value is converted into bytes into a placeholder since an integer is made up of 4 bytes. And thus, the method returns the next free slot in the buffer to convert more data, so that if a struct is made up of an integer and a character they are both put on the same buffer. In lines 83-89 of the deserialization part the same thing is done however this time the bytes are being converted into the original data.

Locks and Join

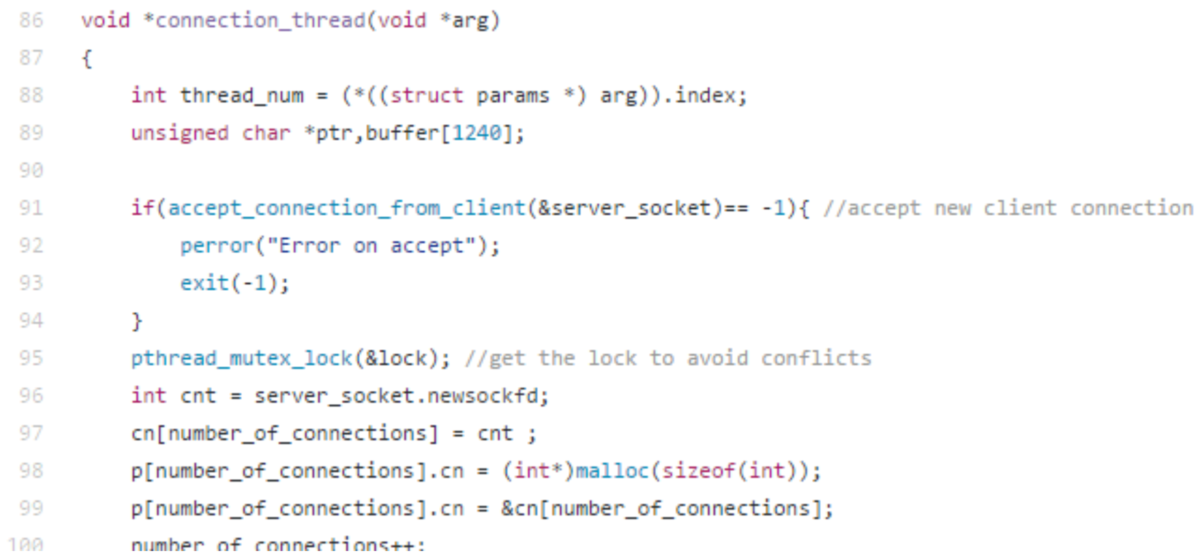
Another mechanism used in the server is that of the mutex locks. The mutex locks are used to prevent conflicts and race conditions. In the current design a conflict can occur whenever a thread that is blocked listening for a connection receives a connection whilst another thread is moving the snakes, this can occur because when a connection is created the number of connections variable is incremented by 1 and a new player is created and hence the server creates threads to move snakes according to how many players are playing. Thus, the server can try to move a snake that is still not initialized and hence can break the game. Thus, as it can be seen below in figure 14 lines 306 and 317, prior to reading and checking for collisions a lock is obtained and then unlocked after, in lines 319 and 310.



```
306         pthread_mutex_lock(&lock); //lock in order to not have conflicts with the connection thread
307         for(int i=1;i<gm->number_of_players+1;i++){ //create threads to get different client input
308             pthread_create(&tids[i], NULL, read_user_input_thread, p + i);
309         }
310         pthread_mutex_unlock(&lock);
311         for(int i = 1; i < gm->number_of_players+1; i++) //wait for threads in order to not check for collisions while a thread
312                                                          //is moving so that to give opportunity to each snake to move first
313         {
314             void *tret;
315             pthread_join(tids[i], &tret);
316         }
317         pthread_mutex_lock(&lock); //lock to not have conflicts with other thread
318         collision(gm); //check for collisions
319         pthread_mutex_unlock(&lock);
```

Figure 14-Obtaining the lock prior to reading and prior to check for collisions

Also for the described mechanism to work, prior to finalize a connection the thread that is waiting for the connection must obtain the lock as shown in figure 15 below line 95:



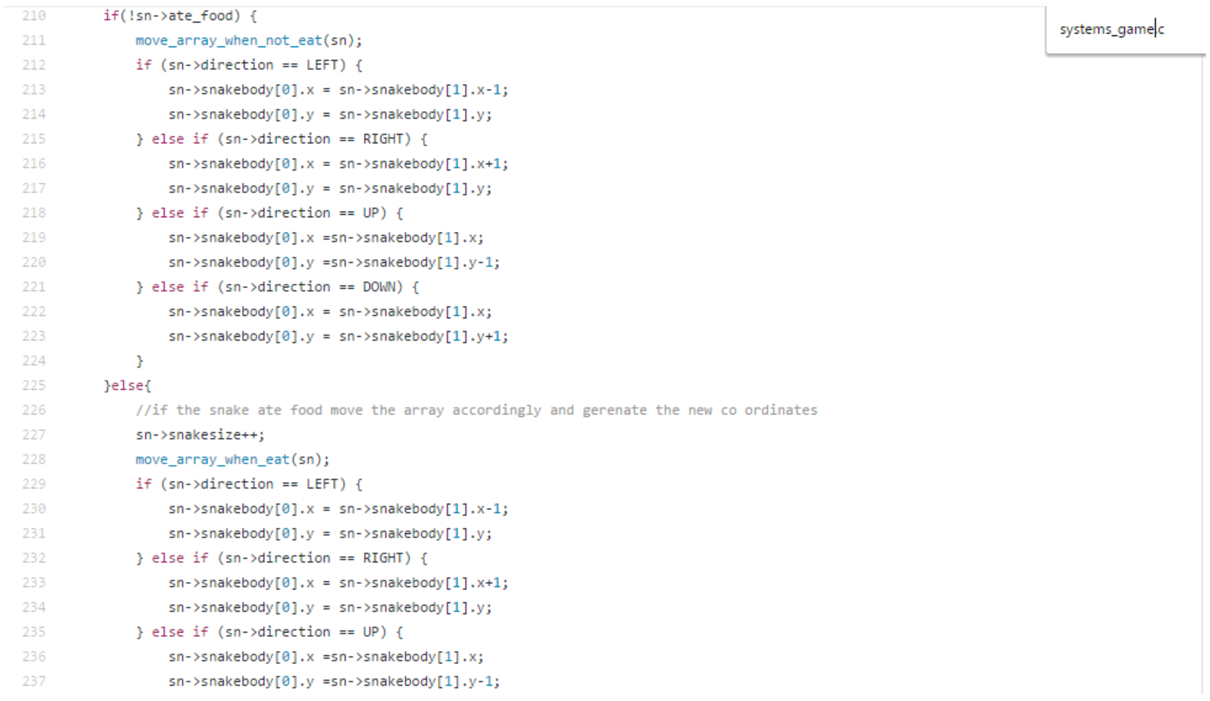
```
86   void *connection_thread(void *arg)
87   {
88       int thread_num = (*((struct params *) arg)).index;
89       unsigned char *ptr,buffer[1240];
90
91       if(accept_connection_from_client(&server_socket)== -1){ //accept new client connection
92           perror("Error on accept");
93           exit(-1);
94       }
95       pthread_mutex_lock(&lock); //get the lock to avoid conflicts
96       int cnt = server_socket.newsockfd;
97       cn[number_of_connections] = cnt ;
98       p[number_of_connections].cn = (int*)malloc(sizeof(int));
99       p[number_of_connections].cn = &cn[number_of_connections];
100      number_of_connections++;
```

Figure 15-Obtaining the lock in the connection thread

Another mechanism used that could be seen in figure 14 above lines 311-314 is to wait for the threads using the join method. This is done so that since different threads are created to perform the read, the main program does not continue to execute without giving the opportunity to all snakes to move once, since if the program starts checking for collisions while a snake has still not moved and another snake has bumped into it could terminate the snake without giving it the opportunity to move and thus it could result into errors in the game. Thus, the join is pivotal in avoiding synchronization problems with the game.

Implementation of the Snake Movement

Since the snake body was implemented to be contained into an array, the movement of the snake could not be done in the best way, but it was done this way to avoid more complications by creating vectors. The snake movement can be seen in figure 16 below.



```
210     if(!sn->ate_food) {
211         move_array_when_not_eat(sn);
212         if (sn->direction == LEFT) {
213             sn->snakebody[0].x = sn->snakebody[1].x-1;
214             sn->snakebody[0].y = sn->snakebody[1].y;
215         } else if (sn->direction == RIGHT) {
216             sn->snakebody[0].x = sn->snakebody[1].x+1;
217             sn->snakebody[0].y = sn->snakebody[1].y;
218         } else if (sn->direction == UP) {
219             sn->snakebody[0].x =sn->snakebody[1].x;
220             sn->snakebody[0].y =sn->snakebody[1].y-1;
221         } else if (sn->direction == DOWN) {
222             sn->snakebody[0].x = sn->snakebody[1].x;
223             sn->snakebody[0].y = sn->snakebody[1].y+1;
224         }
225     }else{
226         //if the snake ate food move the array accordingly and gerenate the new co ordinates
227         sn->snakesize++;
228         move_array_when_eat(sn);
229         if (sn->direction == LEFT) {
230             sn->snakebody[0].x = sn->snakebody[1].x-1;
231             sn->snakebody[0].y = sn->snakebody[1].y;
232         } else if (sn->direction == RIGHT) {
233             sn->snakebody[0].x = sn->snakebody[1].x+1;
234             sn->snakebody[0].y = sn->snakebody[1].y;
235         } else if (sn->direction == UP) {
236             sn->snakebody[0].x =sn->snakebody[1].x;
237             sn->snakebody[0].y =sn->snakebody[1].y-1;
```

Figure 16

As it can be seen above the movement is split into 2, when the snake ate food and when the snake did not eat food. If the snake did not eat food then there is no need to extend the snake body and hence each array element except the last one is shifted to the right in the array for the last element in the snake body to get lost and move forward in the direction and hence put the new head accordingly as shown in lines 212-223.Otherwise, if the snake has eaten food he needs to extend his body and hence as shown in line 227 his body is incremented. Then the snake body's array elements are shifted to the right even the last element since the body is incremented so to create space in the front. Then as shown in lines 229-237 the head is added to the body with coordinates according to the direction of the snake.

Server Decisions

Another design decision was about the server, the server could have been done in a way to never terminate and hence always listens for connections whenever all the players have been disconnected. However, this was done differently as, whenever a game has started and all players have lost the server stops to check that the server ends cleanly. Also, the server never stops if there is a winner each time and thus the game restarts with all those players that still didn’t lose.

For more implementation details see the code listings as there is Dioxygen comments that describe each struct better.

Test Cases & Results

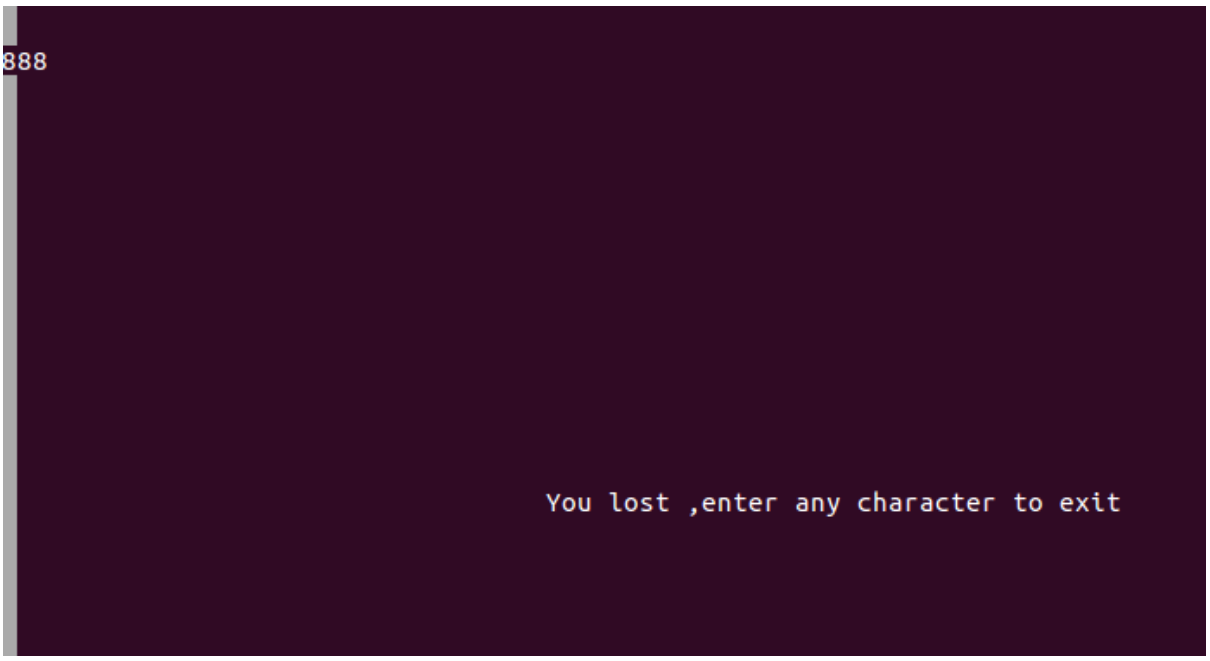
The following tables shows the test cases considered and the respective results, it is important to note that more test cases could be taken but due to time constraints and due to the complexity of the program not all cases could be taken.

Single Player Testing

Test Case 1: Checking that when hitting the arena that the game stops

Expected Output: You lost please enter a character to terminate the program

Real Output:



```
888

                                         You lost ,enter any character to exit
```

Figure 17-Hitting arena termination

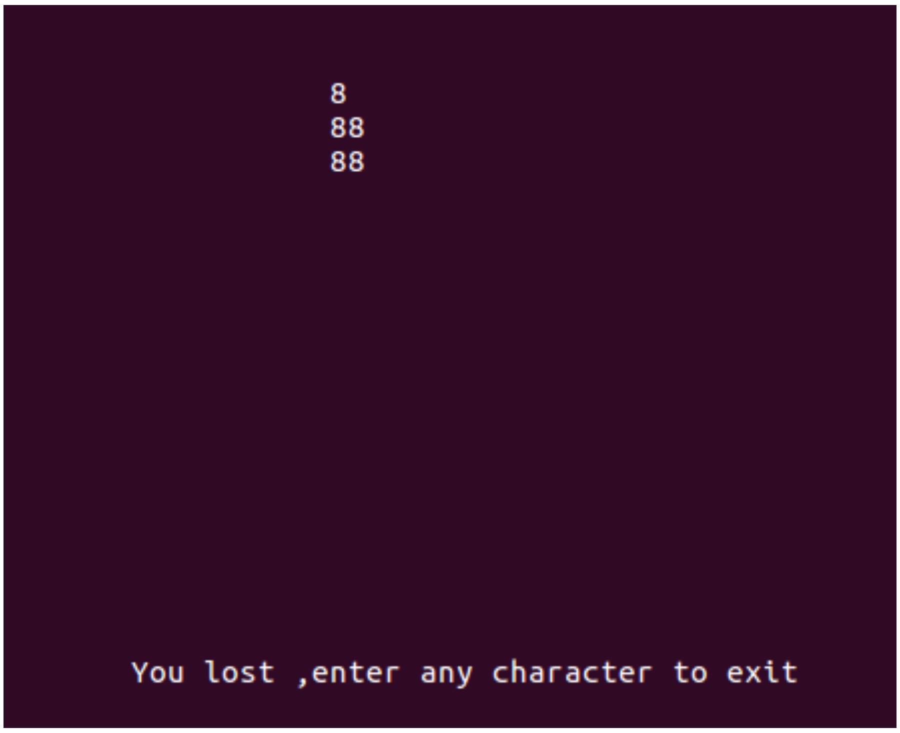
Along with the fact that the server stops

Result: Success

Test Case 2: Checking that when snake collides with himself the game stops

Expected Output: You lost please enter a character to terminate the program

Real Output:



```
8
88
88

You lost ,enter any character to exit
```

Figure 18-Snake hit himself termination

Along with the fact that the server stops

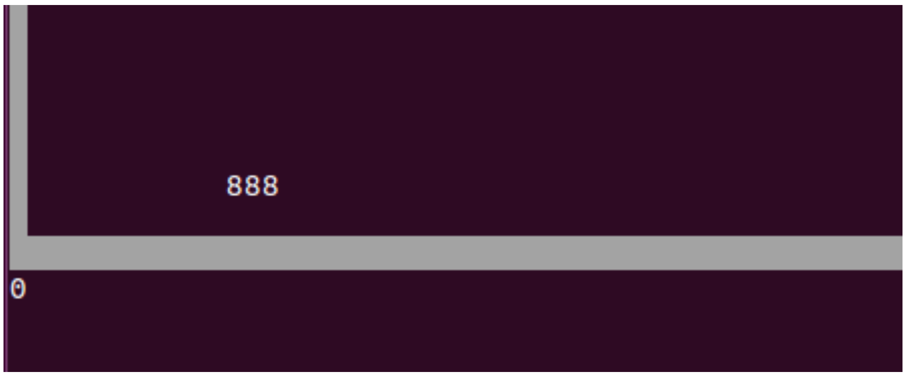
Result: Success

Test Case 3: Checking that when snake eats food his size incremented by 1 and points incremented

Expected Output: Having an initial value of three the snake increments by 1 to size 4 and having 10 as points

Real Output:

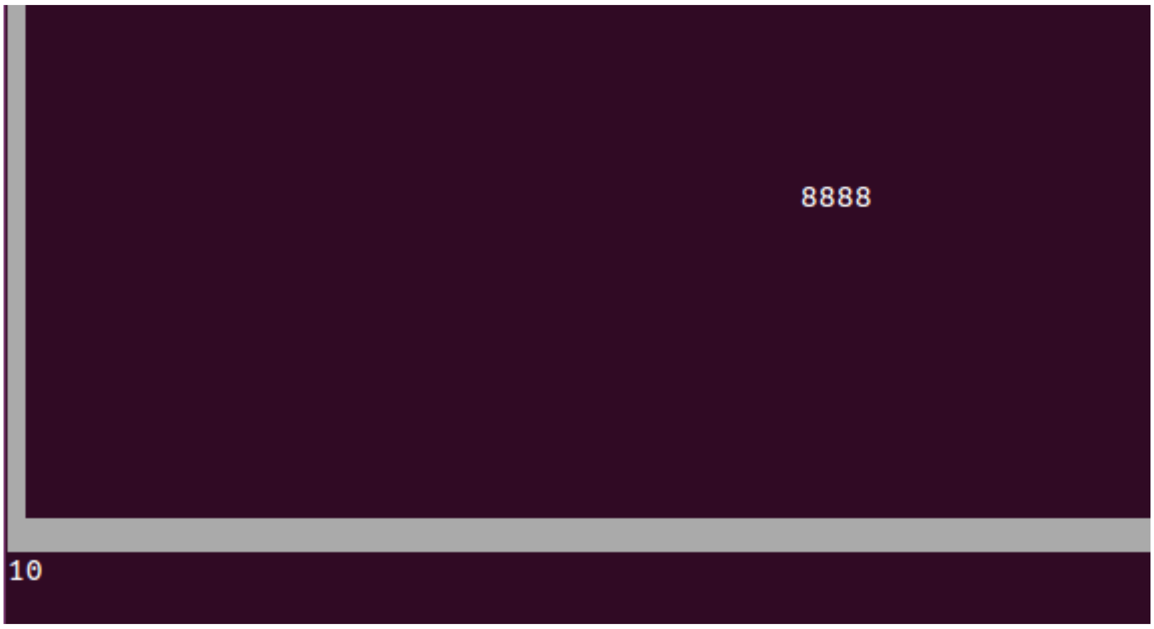
Before:



```
888
0
```

Figure 19-Before eaten food

After:



```
                                          8888




10
```

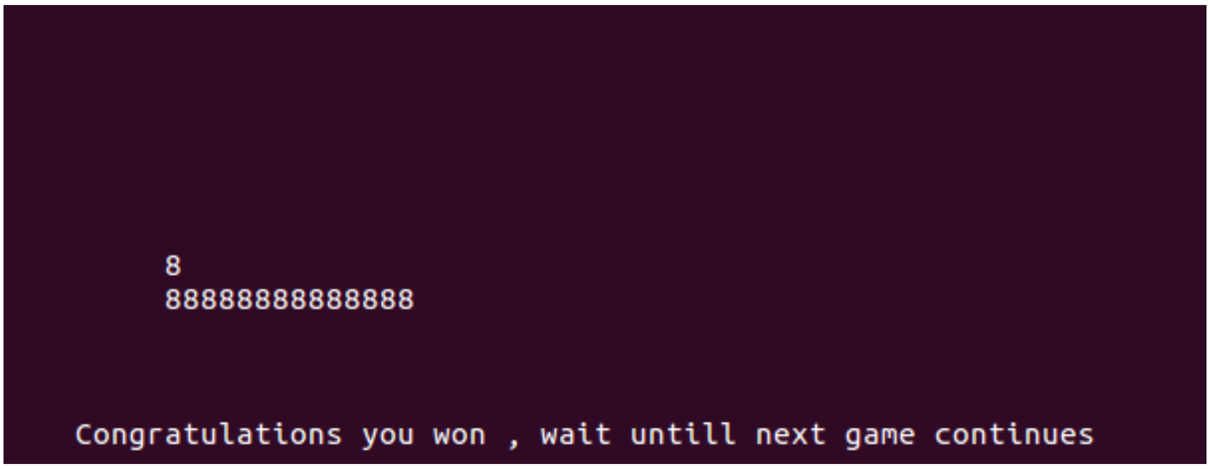
Figure 20-After eaten food

Result: Success

Test Case 4: When winner declared declare him as winner and restart the game with the same snake

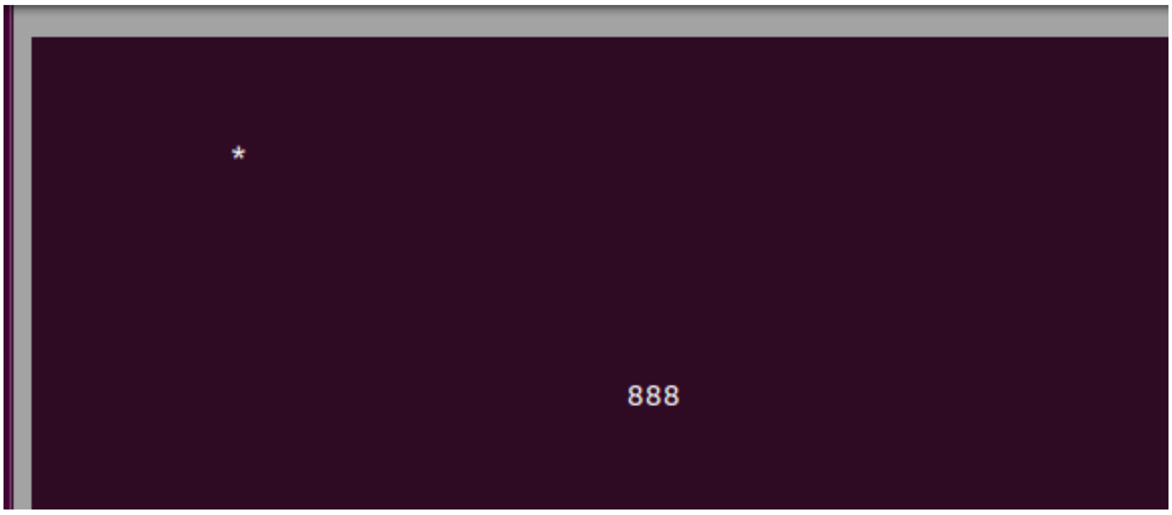
Expected Output: You won wait until game restarts.

Real Output:



```
        8
        88888888888888


    Congratulations you won , wait untill next game continues
```

Figure 21-Terminating game as winner



```
*

                         888
```

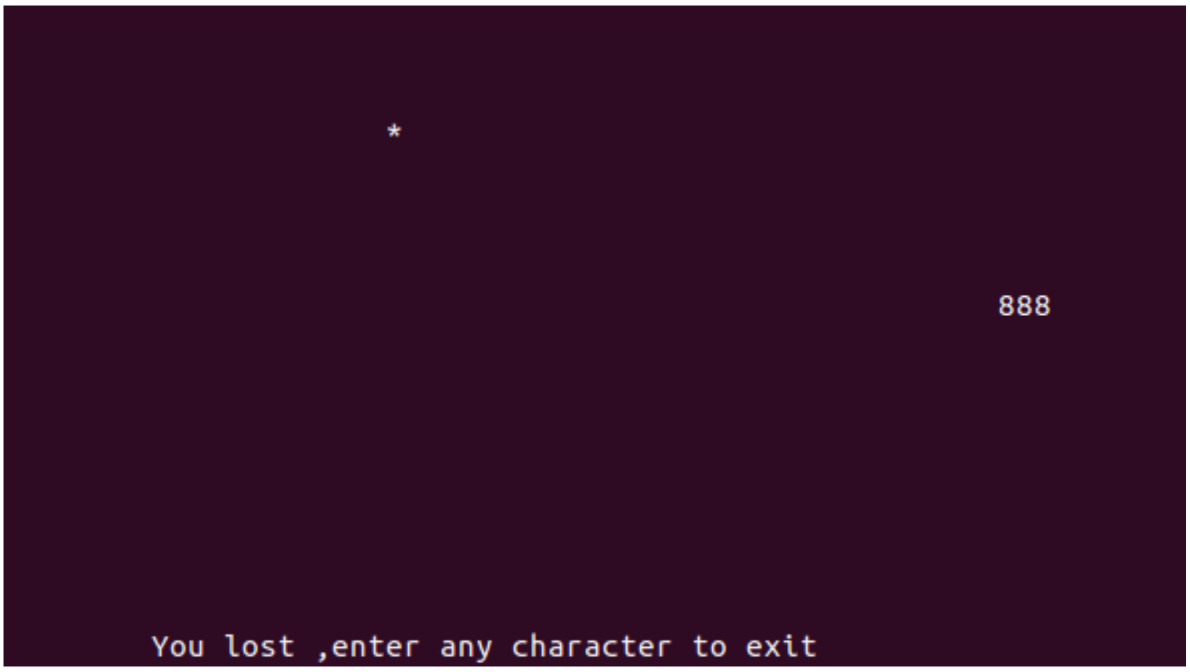
Figure 22-Restarting game again

Result: Success

Test Case 5: Terminating the server and client with CTRL-C

Expected Output: terminating one of them results to both to terminate since the server does not have more clients, and the client needs to stop.

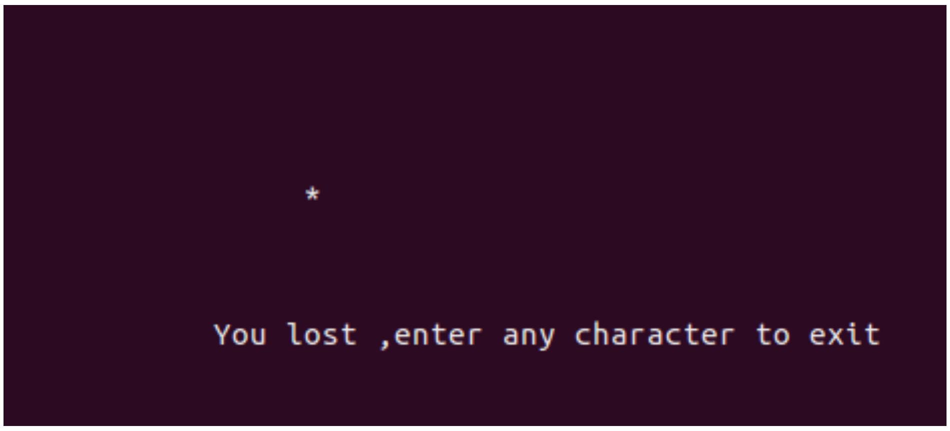
Real Output:



```
        *

                                        888


   You lost ,enter any character to exit
```

Figure 23-After doing SIGINT to the client

Along with the fact that the server stops.



```
          *


               You lost ,enter any character to exit
```

Figure 24-After sending SIGINT to the server

Along with the fact that the server stops.

Result: Success

Thus, the single player snake is supposed to work fine.

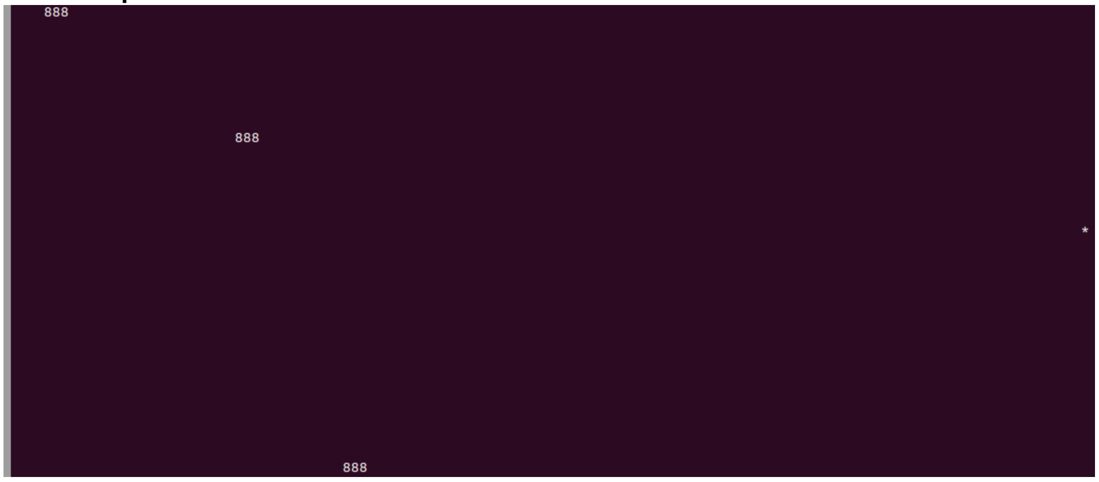
Multiplayer Testing

The Multiplayer testing was tested on the same machine for some test cases because it was impossible to test the game by the same person on different machines, however as it can be seen in the last test case the connection over different machines was tested to check that the game starts over different machines. The game was not tested on different machines for all test cases because if the game logic is correct on the same machine it must be correct on different machines also. Also, the multiplayer testing was carried out using 3 clients

Test Case 1: Checking that the game starts correctly with 3 clients and hence connection is good

Expected Output: 3 Snakes moving on terminal

Real Output:



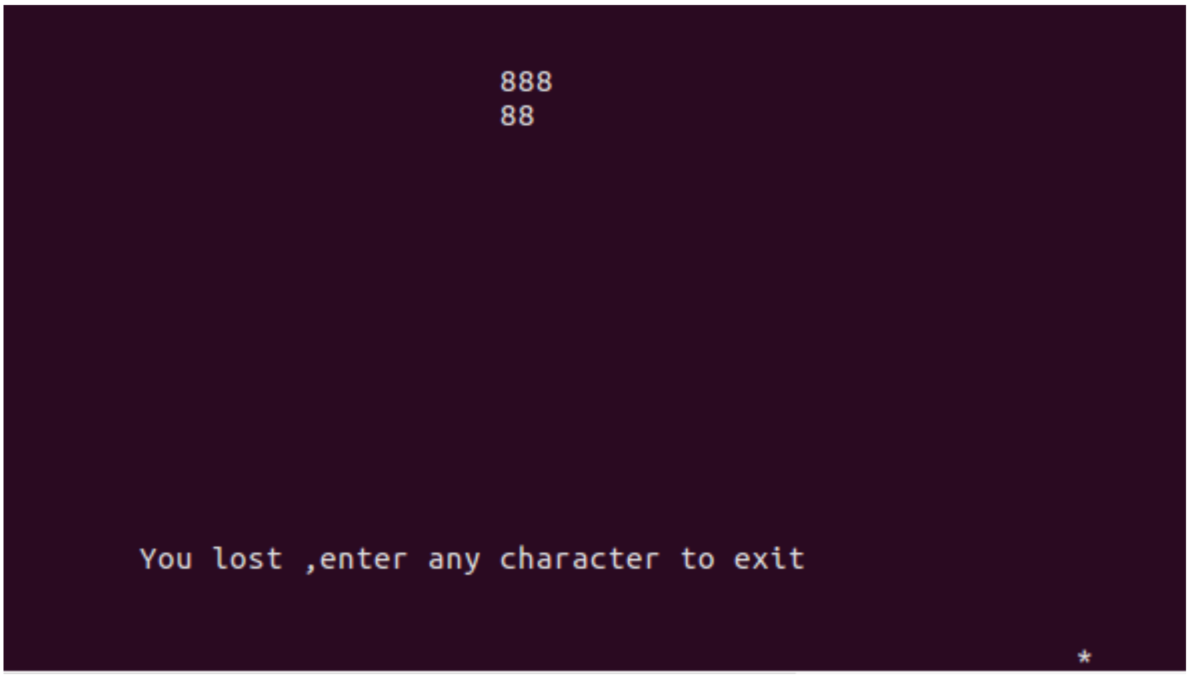
```
  888

               888
                                                                            *




                          888
```

Result: Success

Test Case 2: Checking if 2 snakes hit head on those snakes are terminated and hence the other snake keeps on playing

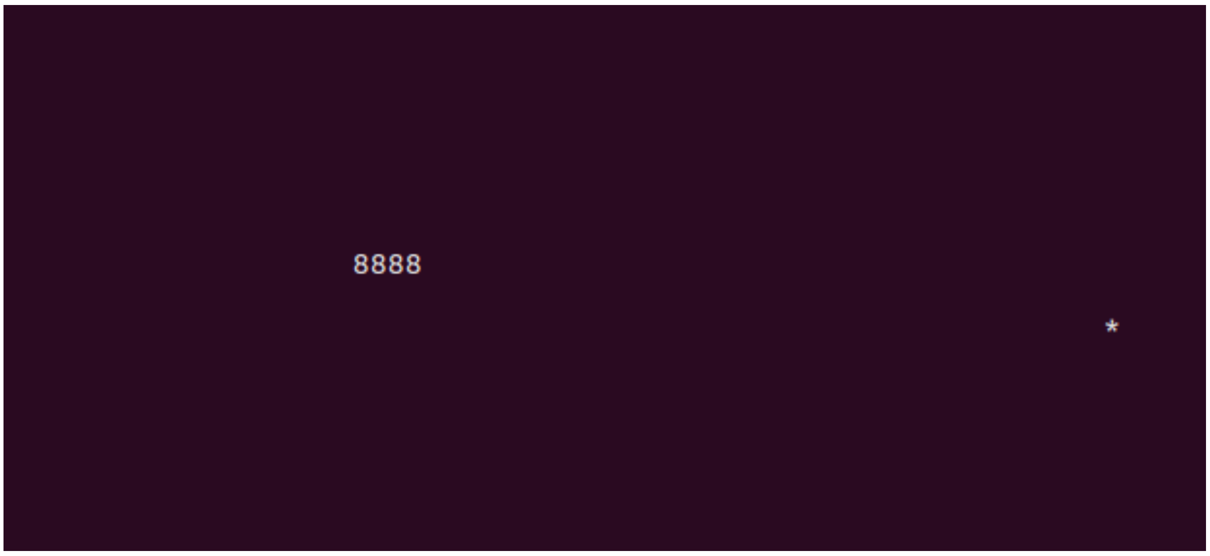
Expected Output: For the killed clients inform them that they lost, for the other player he should keep on playing and thus the server does not terminate.

Real Output:



```
                              888
                              88





      You lost ,enter any character to exit

                                                          *
```

Figure 25-Client 1 and 2 after hitting head on



```
                  8888
                                                            *
```

Figure 26-Client 3 kept on playing

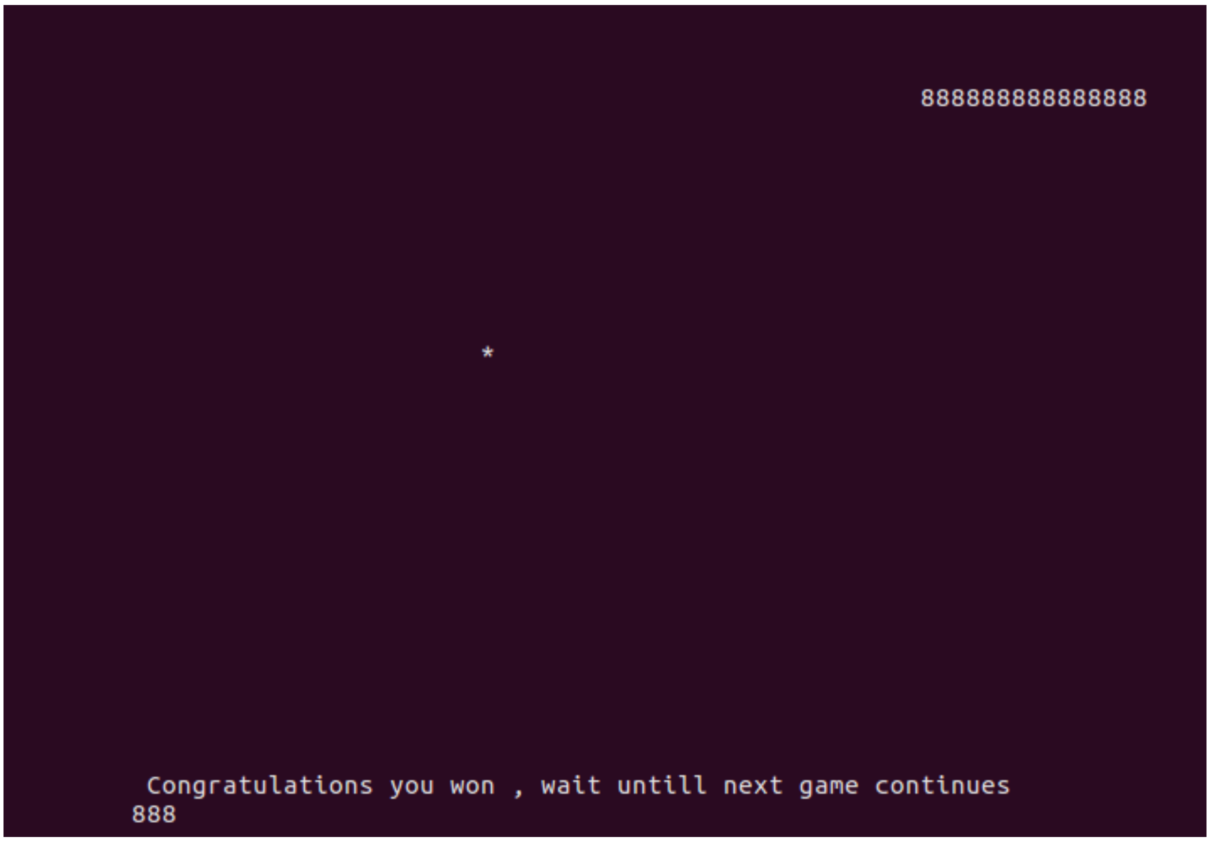
Along with the fact that the server does not terminate.

Result: Success

Test Case 3: Checking if a snake is killed and 2 clients are still playing were one is a winner the game restarts with those 2 players

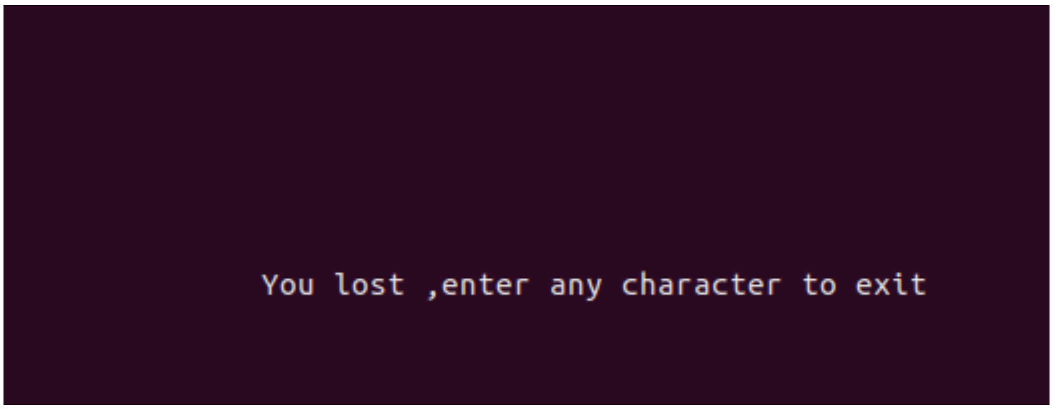
Expected Output: The one that collides will lose, the other is declared as winner and game restarts with the 2 snakes

Real Output:



```
                                                            888888888888888




                               *





         Congratulations you won , wait untill next game continues
        888
```

Figure 27-client's 1 terminal after winning



```
            You lost ,enter any character to exit
```

Figure 28-client 2's terminal after losing, hence game did not restart with him

Figure 29-game restarts again with the 2 snakes

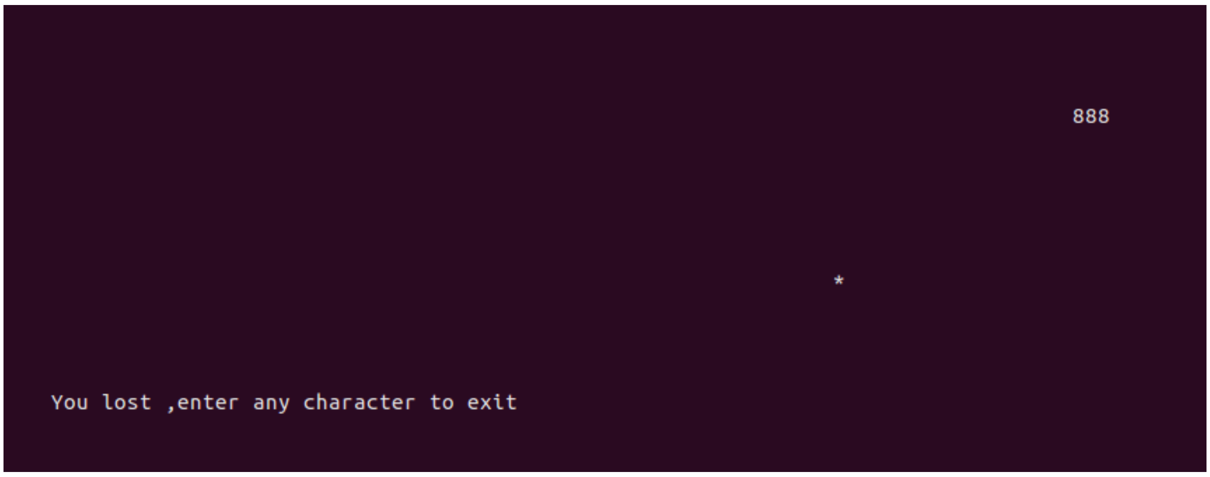
Along with the fact that the server does not terminate.

Result: Success

Test Case 3: When a snake hits another snake not in the head

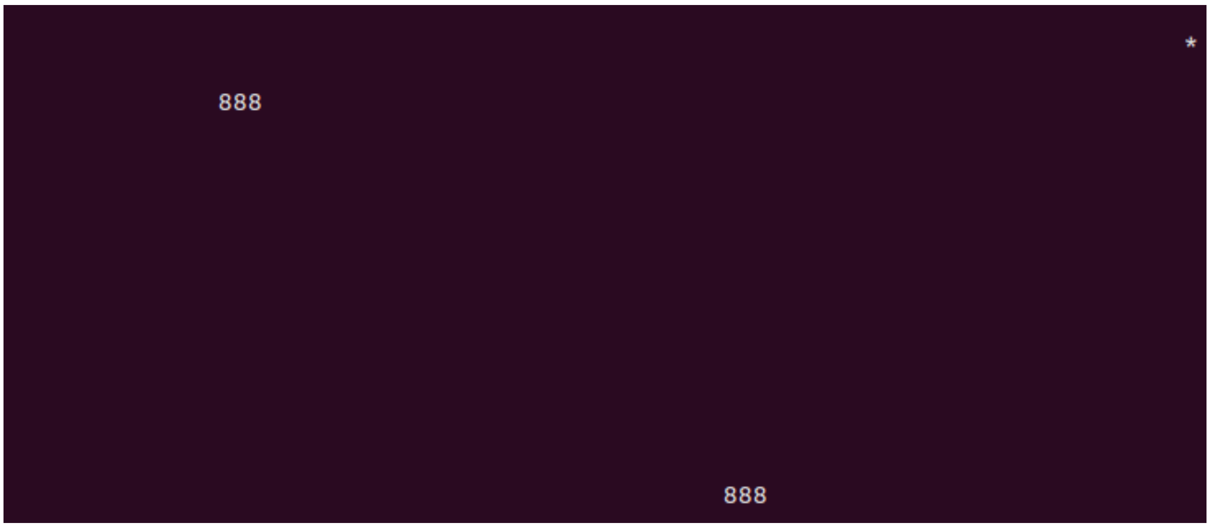
Expected Output: The one that collides will lose and the other 2 must keep on playing

Real Output:



```
                                                            888



                                               *



  You lost ,enter any character to exit
```

Figure 30-client 1 lost because he lost



```
                                                                         *
          888









                                      888
```

Figure 31-other 2 clients kept on playing

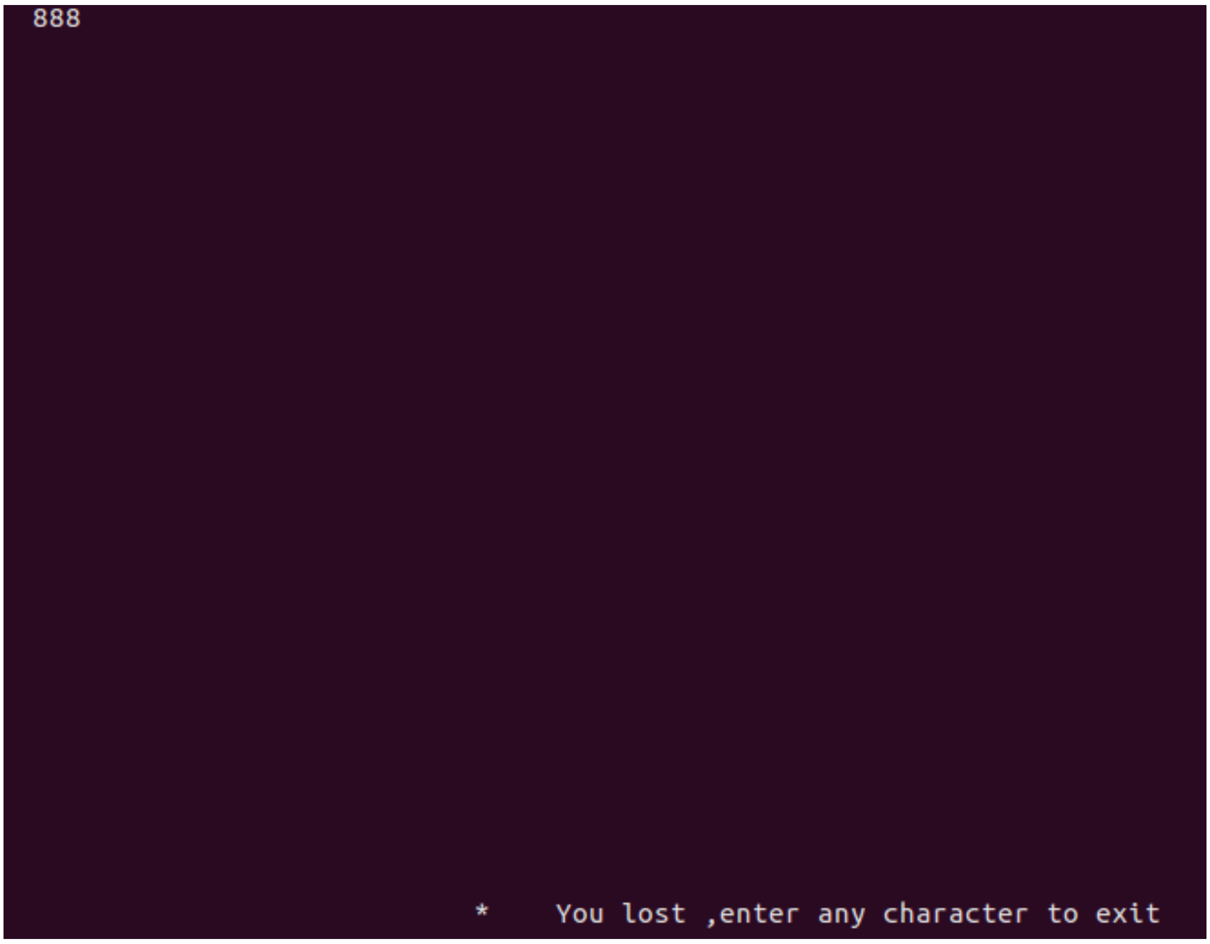
Along with the fact that the server does not terminate.

Result: Success

Test Case 4: Sending SIGINT to Server or Client

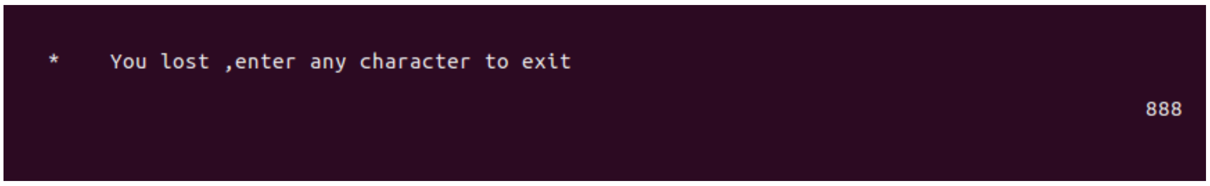
Expected Output: Sending SIGINT to the server all clients must terminate, sending it to the client only that client must be terminated

Real Output:



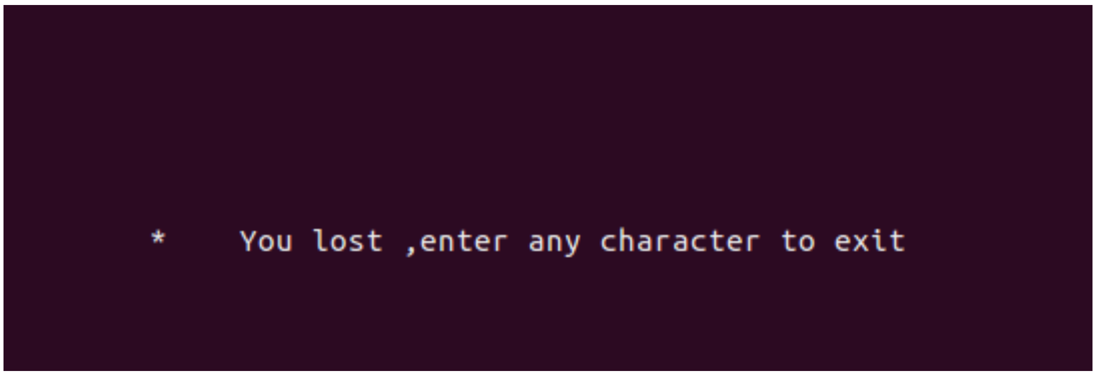
```
888




                              *    You lost ,enter any character to exit
```

Figure 32-Client 1 after sending SIGINT to the server



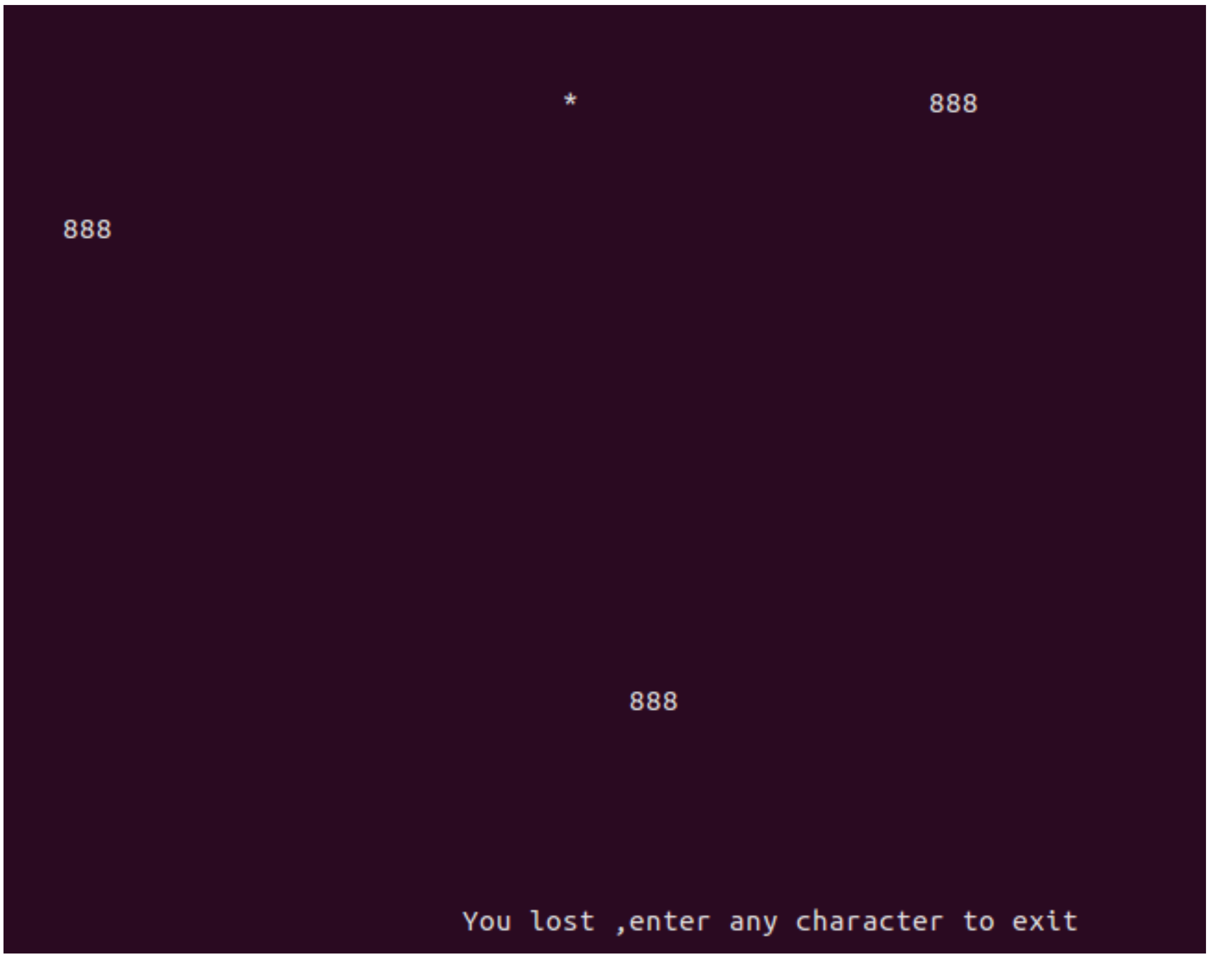
```
  *    You lost ,enter any character to exit
                                                                  888
```

Figure 33-Client 2 after sending SIGINT to the server



```
      *    You lost ,enter any character to exit
```

Figure 34-Client 3 when sending SIGINT to the server



```
                         *                  888

  888




                            888



                    You lost ,enter any character to exit
```

Figure 35-Client 3 after sending SIGINT to client 3

```
                                        888



888
```

Figure 36-Other clients do not terminate

Along with the fact that the server does not terminate

Result: Success

Test Case 5: Checking that over different machines the game works and all the above test cases work fine. For this test case, no output is given since the same output is given as in the separate test cases above.

Expected Output: The game is seen to be working as expected

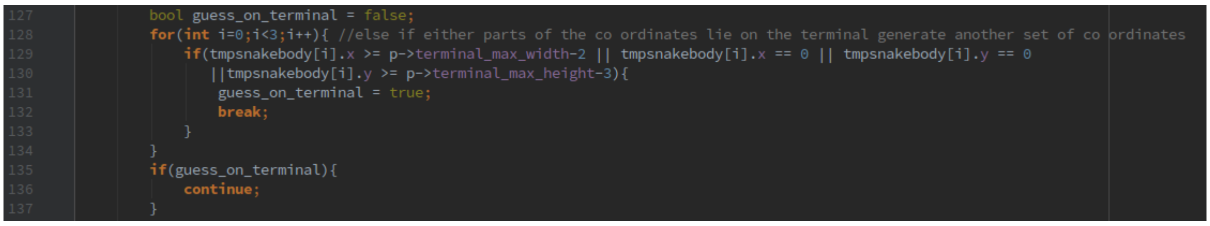
Real Output: The game is seen to be working as expected

Result: Success

List of Improvements & Bugs

Bugs

The game requirements work correctly however there are some bugs that are present in the solution. One of the bugs is that the number of players that can play the game is set to be 400, although this is a very large number the requirement in the assignment specification was that the game should not have a limited number of players, but since the list of players is contained in an array it must be of fixed size. Also, for the terminal screen to not be overcrowded a limit had to be set otherwise the game could be broken. Another bug in the game is that sometimes the snake is started with the head in the arena and hence the client is terminated immediately, this issue could not be determined on why this happens because as it can be seen in figure 25 below lines 129-136, the terminal border is checked and hence that generated snake positions should be generated again until they are correct.



```
127         bool guess_on_terminal = false;
128         for(int i=0;i<3;i++){ //else if either parts of the co ordinates lie on the terminal generate another set of co ordinates
129             if(tmpsnakebody[i].x >= p->terminal_max_width-2 || tmpsnakebody[i].x == 0 || tmpsnakebody[i].y == 0
130                ||tmpsnakebody[i].y >= p->terminal_max_height-3){
131                 guess_on_terminal = true;
132                 break;
133             }
134         }
135         if(guess_on_terminal){
136             continue;
137         }
```

Figure 37

The last bug encountered in the game is that the game must be played with a full screen terminal size, otherwise the game could not be displayed fully, however in the client the wresize function is used so that the game does not break. To tweak the terminal size, modify that parameters passed to the wresize function. One must also note if the terminal size is smaller than the wresize function characters then even if maximizing the game is not displayed to the full, hence the arguments should be modified.

Another bug is that when the killed function is used this cannot be handled correctly since according to the Linux man pages the kill signals do not terminate cleanly and hence no handlers are invocated, thus only the SIGINT signal is handled by the server and client.

There could be also race conditions bugs or synchronization bugs due to latency, however these were not met during the testing.

One must also add that in this game whenever a new client joins the game, the game is stopped for 2 seconds for the clients to know what will happen and where will the new snake be put in the game. Also, after the game executes, a few minutes should be given to prevent the Error-Binding error.

Improvements

Although the game solution is fine there could be further improvements to the game such as to make the server never terminate. The server could be done to always wait for connections even if there are no clients playing a game, however as previously described, this was done in this way for simplicity and to show that the server closes cleanly.

Another improvement could be to use shared memory or another structure between related processes on the same machine to improve latency and hence do not require always the use of a TCP connection, however this was not done this way to not make the solution more complex and since more complex IPC structures are needed since more deadlocks and race conditions could occur.

Another improvement could be to use graphics for the terminal to look nicer.

Conclusion

To conclude, the assignment requirements were all met however as described in the previous section some mechanisms could be done better with more mechanisms to avoid race conditions and synchronization problems. Also, the goals set in the beginning of the assignment such as to learn about the connectivity protocols using TCP/IP, the goals of multithreading and the use of locks were all achieved because they were implemented successfully in the solution and thus creating a working solution which is efficient at the same time.

As also, described in the last section although the solution works, there could be further improvements in the game graphics, use of IPC structures for related processes to avoid race conditions and synchronization problems to improve latency of the TCP/IP connection.

References

[1] ***A Brief History of C***. Available: https://www.le.ac.uk/users/rjm1/cotter/page_06.htm.

[2] ***What is Systems Programming?***. Available: http://cs.lmu.edu/~ray/notes/sysprog/.

[3] (September 24, 2007). ***How important is multithreading in application development?***. Available: http://www.techrepublic.com/blog/software-engineer/how-important-is-multithreading-in-application-development/.

[4] Available: http://man7.org/linux/man-pages/man1/kill.1.html.

[5] (May 2012). *Introduction to Sockets Programming in C using TCP/IP*. Available: http://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf.

[6] ***Chapter 1. Introduction to TCP and Sockets***. Available: https://www.scottklement.com/rpg/socktut/introduction.html.