

GROUP PROJECT 2

HAUNTED

BIANCA EBANKS, JACOB DEMEY,
KRZYSZTOF KOCHANCZYK, LOGAN MCNEELY,
MOHAMMED ALYAHYA, OMAR ALMOTAIRI
ITEC 2085
TOM CALABRESE
16 MAY 2018

CONTENTS PAGE

Problem Description	2
Gather Data	4
Comprehensive Review of Underlying Network Technologies	5
Functional Application Design Narrative	6
Flow of Finite State Diagram	7
Message Types	8
IPO Chart	11
Client	13
Server	14
Flow Chart	15
Receiving	15
Parsing (server & client)	15
Registration	16
Accepting User Input	16
Creating Message	17
Creating Error Message	17
Sending	18
Moving Figures	18
Displaying Board	19
Checking IP	19
Ending Game	20
Implementation	21
Client	21
Server	24
Testing	39
Network:	39
Generalization	44
Group Assessment	45

Problem Description

Multiplayer Online Haunted Game

The team is required to execute the design process to create a valid design to facilitate the implementation of the Multiplayer Online Haunted Game which is very similar to a traditional game of Pac-Man. The systems may exist on a variety of subnets or on the same LAN.

The game will take place on the traditional Pac-Man board. The game is played by people who have joined the game by registering with the server. Once a single player is registered the game will start. The controls are: W is up, A is left, S is down, and D is right, which are traditional gaming keys. Players may join in at any time and continue to play, the game takes all inputs and sets them in a queue and will play every move unless the player runs into a ghost, which will end the game immediately. The ghosts all start in the middle of the screen and all move in a random direction everytime the player makes a move.

The objective of the game is to attempt to gain as high of a score as possible. This can be done by working together with all the players and avoiding the ghosts, instead of just inputting like a mad house and not being able to see the screen.

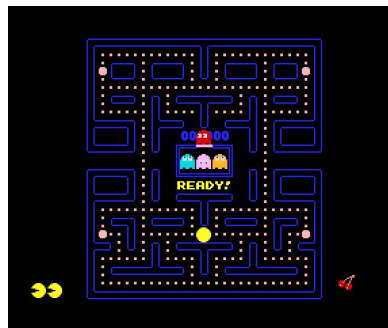


Figure 1. Traditional Pac-Man board

Existing State:

Does not exist

Desired State:

To have from 1 - 5 computers connected to a single server that the client will then play a game of Haunted. The server will be the single display of the board and each player will have a decision on which direction the “man” moves at a any time.

Criteria	Constraints
Update players location	“Limited” to 5 players
Add players during game	Server is the only device that will show/update the GUI therefore players need to be within sight of it.
Close game whenever the player moves into a ghost	Constant broadcasting of data between server and client
Mad house style of play (All moves are placed in a queue and all are executed in order of input)	Firewall prohibits connections - need to whitelist or disable before playing game.
Players will score points as they collect dots around the map.	

Gather Data

- Libraries we needed to look into:
 - <winsock2.h> - For sending and receiving data over a socket
 - <conio.h> - For MSDOS input/output
 - <windows.h> - A Windows specific header that contains functions to the Window's API
 - <stdbool.h> - A library that adds boolean values true and false to C/C++
 - <ctime> - A library used to manage time within the program
- Programs :
 - Wireshark - To track and troubleshoot networking packets sent/received

Comprehensive Review of Underlying Network Technologies

~26 Windows 10 PCs

LAN/WLAN

Ethernet-Connected

Johnson & Wales Network

Wireless access point (for additional devices) - These devices use a 10.129.X.X format for their IP addresses.

Functional Application Design Narrative

	Final Choices	Reason
TCP vs. UDP	UDP	Doesn't matter if a few packets are lost, just keep updating
Concurrent vs. Iterative	Iterative	Client will be changing the game depending on their input
Stateful vs. Stateless	Stateful	Have to remember information from every client
Ports sockets vs RPC vs Message passing	Ports and Sockets	All the computers are in the same room
Architecture	Client-Server	The server is running the game, the rest are clients that control the movement of the man in the game
Available vs. Consistent	Available	Every request should succeed and receive a response
Data Protection	None	No need for data protection since we are iterative

Flow of Finite State Diagram

The client comes out of the idling stage upon “start up” and attempts to connect to the server. In the event that a connection could not be established the client will create an error message and proceed to attempt recovery until a successful connection is made or the client gives up. If the connection is successfully made, the client will receive a message asking if they would like to register for the game. If they would like to register they will be sending a ‘Y’ which is the M1 message to the server.

Depending on the message sent, the server can complete one of five actions. If message m1 was sent then the server will register the client and update the IP table on the server side. The server will then send a message back to the client with instructions to input a direction, begin the game and display the playfield. The client will then receive a message and parse it, in this case the message would still be m1 so it would try to register again and realize it already has, so the client will be asked to create a message (M2, M3, M4 or M5) that will be sent over to the server.

The server will then again receive and parse through the information from the message that was sent. Whichever message that was sent by the client, determines what the server will do. If the client sends an M2 message the server will move the man one spot UP, M3 will move the man one spot to the LEFT, M4 will move the man one spot DOWN, and M5 will move the man one spot to the RIGHT.

After the man’s direction is updated the client will get a successfully delivered and executed message. The server will then ask the client for a new direction to move the man. This will continue until the man is either eaten by a ghost, or has eaten every one of the dots. If the man is eaten, the client will receive a “YOU LOSE!” message and their game will be over.

Message Types

M1: Register

M2: Move Up

M3: Move Left

M4: Move Down

M5: Move Right

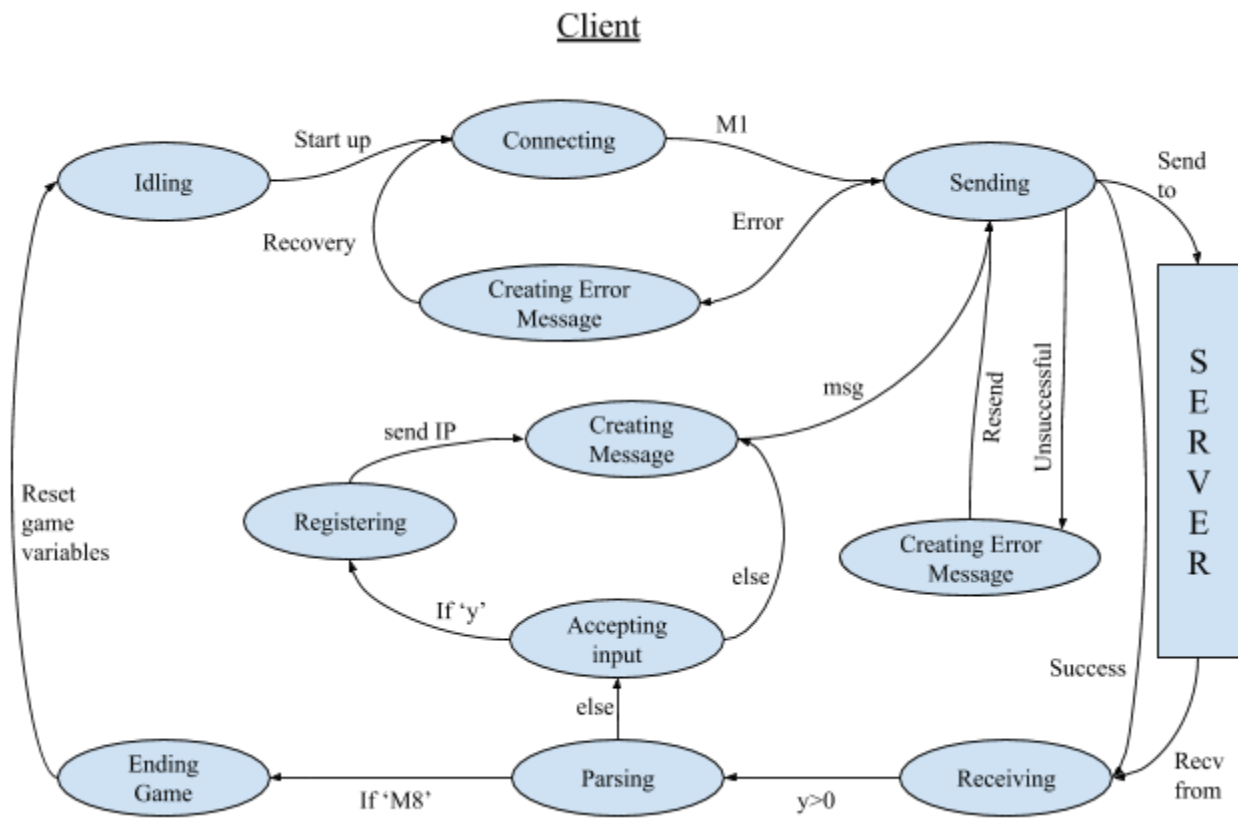
M7: Not registered

M8: You Lose

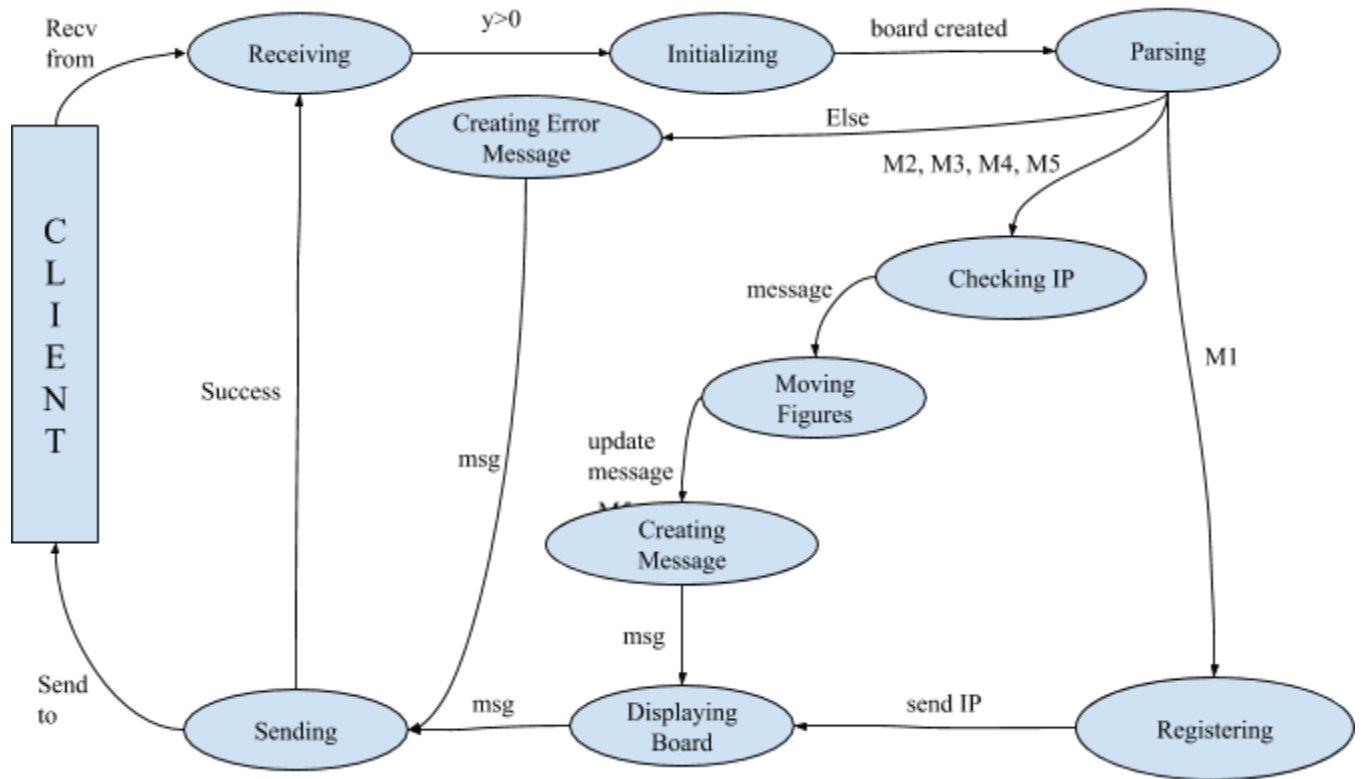
M9: Wrong Way

M1	IP (int) 4	register (int) 4
M2	IP (int) 4	up (int) 4
M3	IP (int) 4	left (int) 4
M4	IP (int) 4	down (int) 4
M5	IP (int) 4	right (int) 4
M7	IP (int) 4	Not registered(char[15]) 15
M8	You lose (char[8]) 8	
M9	Wrong way(char[9]) 9	

Finite State Diagrams

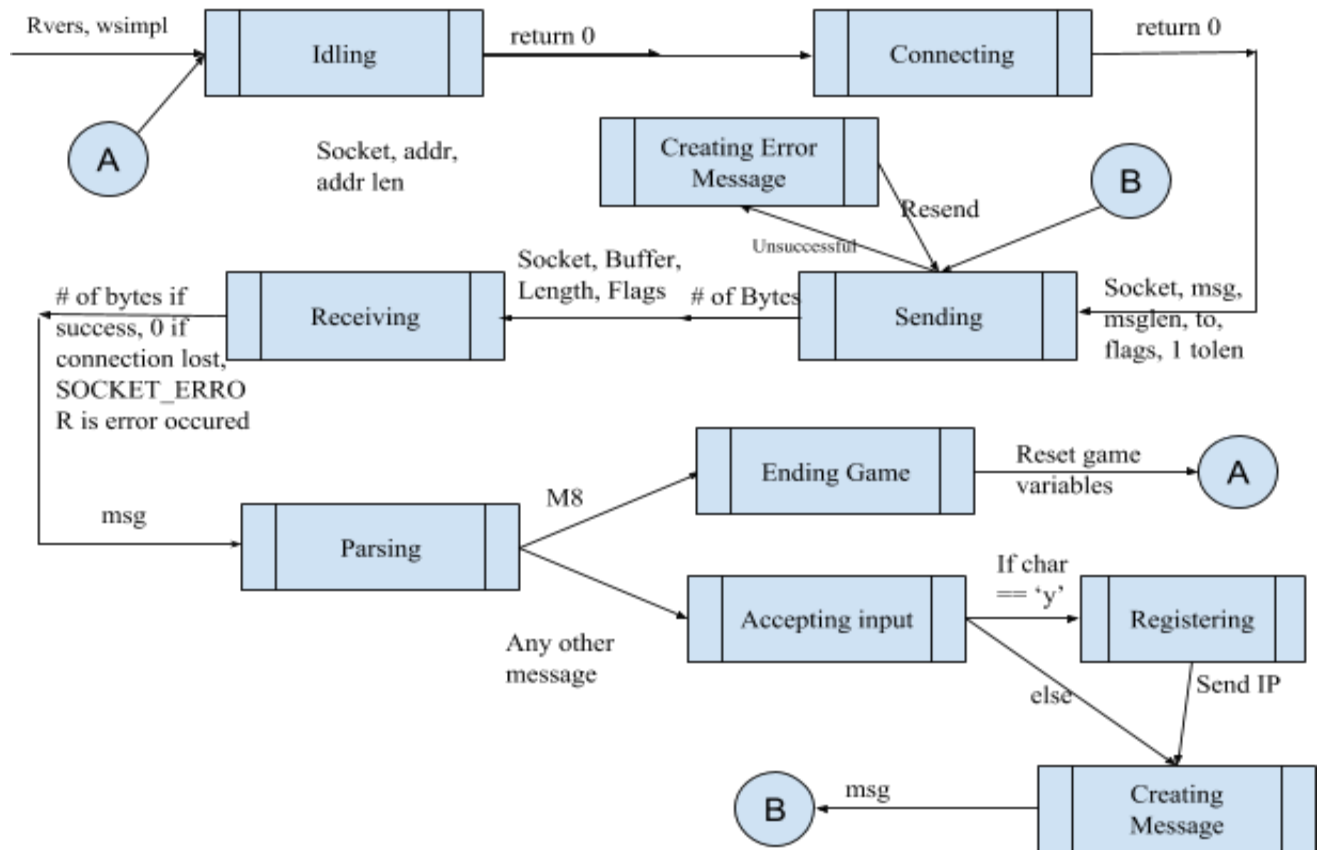


Server

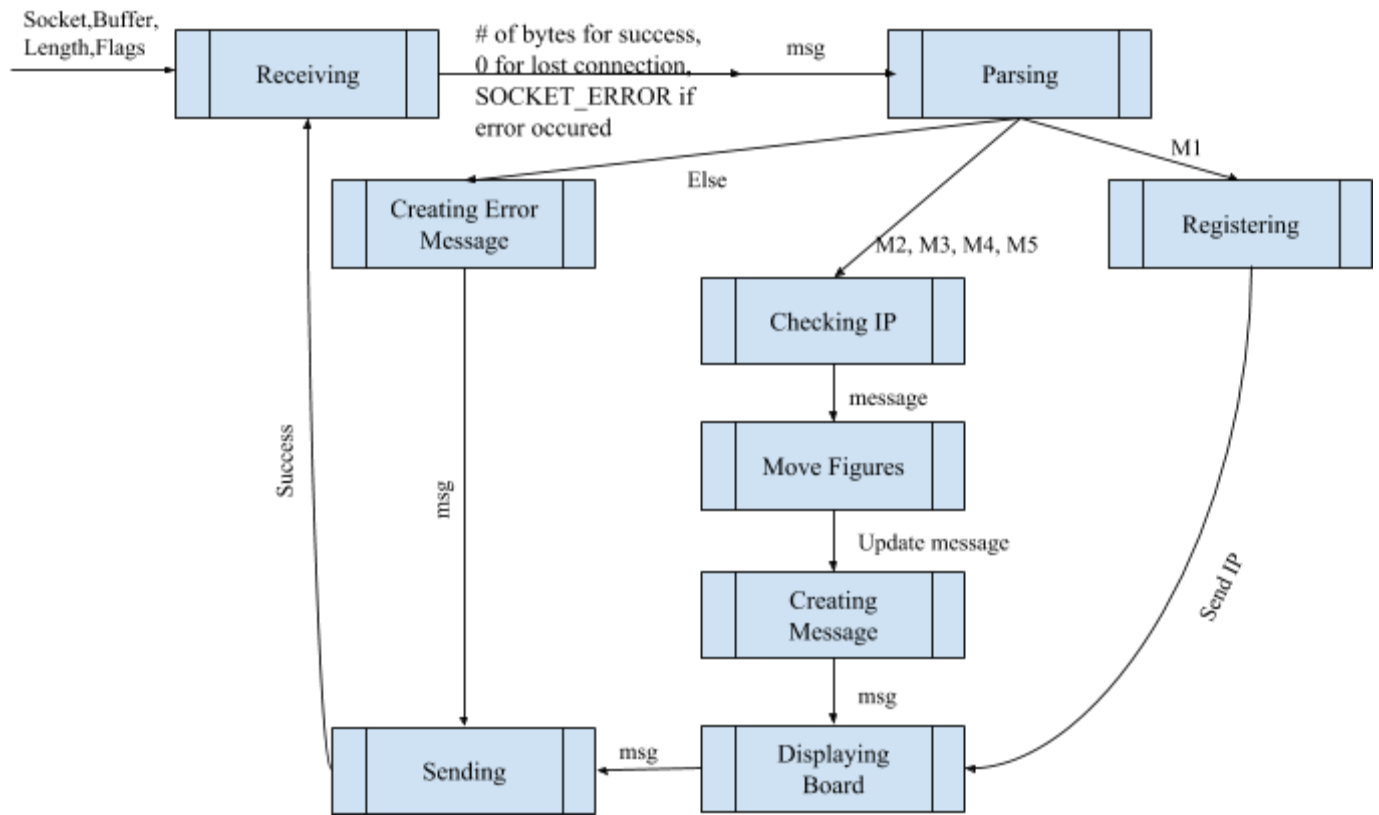


IPO Chart

Client



Server



Transition Matrix

Client

	M 1	M 2	M 3	M 4	M 5	M 7	M 8	M 9	M	A	B	C	D	E	F	G
1.	2.	2.	2.	2.	2.	2.	2.	2.	2.	11.	11.	11.	11.	11.	11.	11.
2.	3.	3.	3.	3.	3.	3.	10.	3.	3.	11.	11.	11.	11.	11.	11.	3./10.
3.	4.	5.	5.	5.	5.	5.	11.	5.	5.	11.	11.	11.	11.	11.	11.	11.
4.	5.	11.	11.	11.	11.	11.	11.	11.	11.	11.	11.	5.	11.	11.	11.	11.
5.	9.	9.	9.	9.	9.	9.	11.	9.	9.	11.	11.	9.	11.	11.	11.	11.
6.	11.	11.	11.	11.	11.	11.	7.	11.	11.	7.	11.	11.	11.	11.	11.	11.
7.	11.	11.	11.	11.	11.	11.	9.	11.	11.	11.	9.	11.	9.	11.	11.	11.
8.	11.	11.	11.	11.	11.	11.	11.	11.	11.	11.	7.	11.	7.	11.	9.	11.
9.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.	8.	1.	1.	1.	8.	1.
10.	11.	11.	11.	11.	11.	11.	6.	11.	11.	11.	11.	11.	11.	11.	11.	11.
11.	9.	9.	9.	9.	9.	9.	9.	9.	9.	9.	9.	9.	9.	9.	9.	9.

1. Receiving

2. Parsing

3. Accepting input

4. Registering

5. Creating message

6. Idling

7. Connecting

8. Creating error message

9. Sending

10. Ending game

M1. 'Y'

M2. 'W'

M3. 'A'

M4. 'S'

M5. 'D'

M7. "You are not registered"

M8. "YOU LOSE!"

M9. "WRONG WAY!"

M. "Invalid Entry!"

A. Start up

B. Error

C. Send IP

D. Recovery

E. Success

F. Unsuccessful

G. Y>0

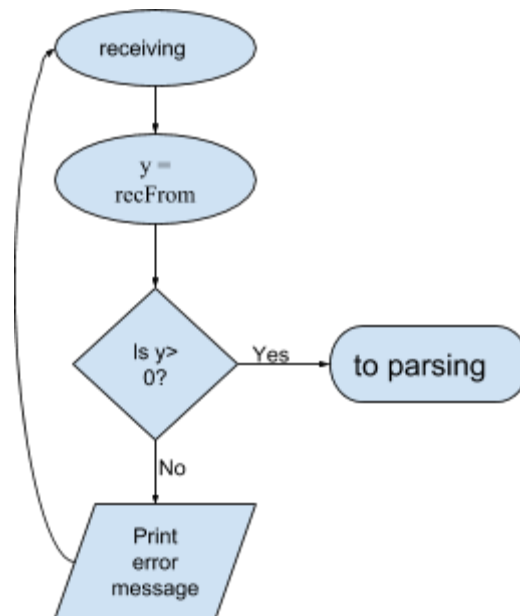
Server

	M1	M2	M3	M4	M5	A.	B.	C.	D.	E.
1.	2.	3.	3.	3.	3.	11.	11.	11.	2.	11.
2.	3.	3.	3.	3.	3.	3.	11.	11.	11.	11.
3.	5.	6.	6.	6.	6.	11.	5.	11.	11.	11.
4.	10.	10.	10.	10.	10.	11.	11.	11.	11.	11.
5.	9.	11.	11.	11.	11.	11.	11.	11.	11.	11.
6.	11.	7.	7.	7.	7.	11.	11.	11.	11.	11.
7.	11.	8.	8.	8.	8.	11.	11.	11.	11.	11.
8.	11.	9.	9.	9.	9.	11.	11.	9.	11.	11.
9.	10.	10.	10.	10.	10.	11.	11.	11.	11.	11.
10.	1.	1.	1.	1.	1.	1.	1.	1.	1.	1.
11.	10.	10.	10.	10.	10.	10.	10.	10.	10.	10.

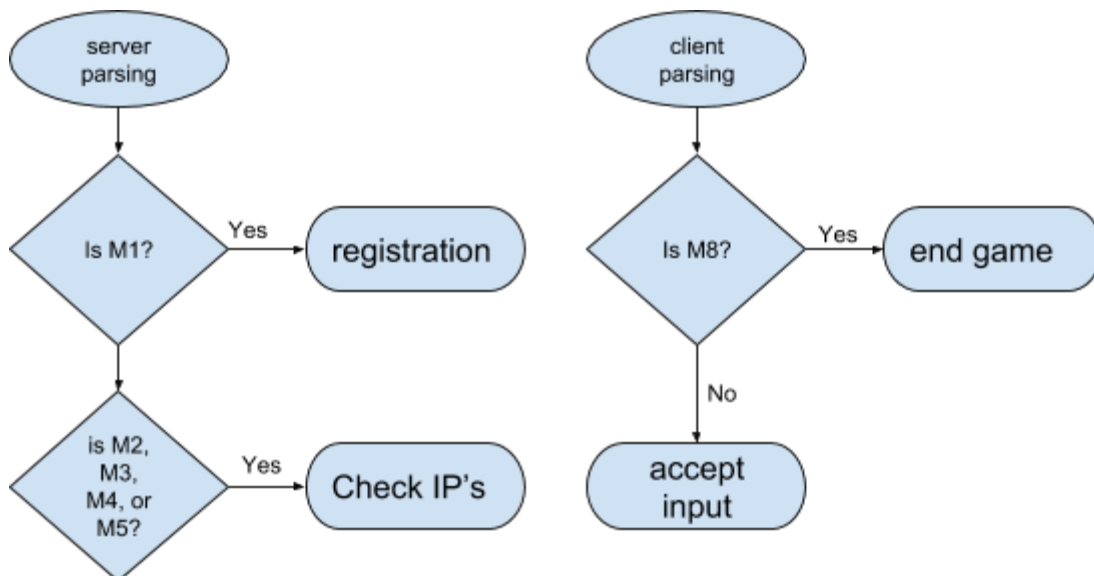
- | | | |
|---------------------------|---------|-------------------|
| 1. Receiving | M1. 'Y' | A. $y > 0$ |
| 2. Initializing | M2. 'W' | B. Board Created |
| 3. Parsing | M3. 'A' | C. Update Message |
| 4. Creating Error Message | M4. 'S' | D. Success |
| 5. Registering | M5. 'D' | E. Unsuccessful |
| 6. Checking IP | | |
| 7. Moving Figures | | |
| 8. Creating Message | | |
| 9. Displaying Board | | |
| 10. Sending | | |
| 11. Error | | |

Flow Chart

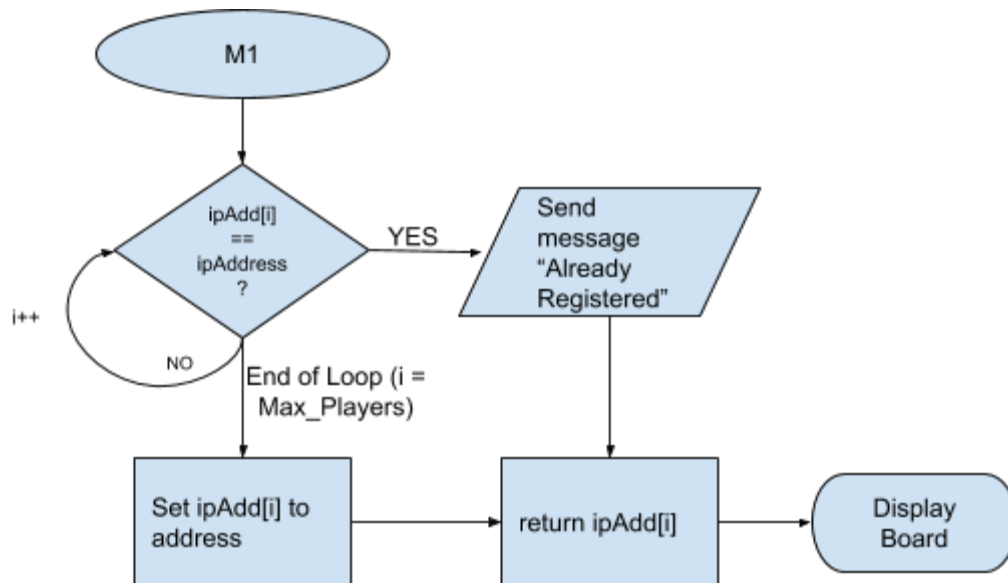
Receiving



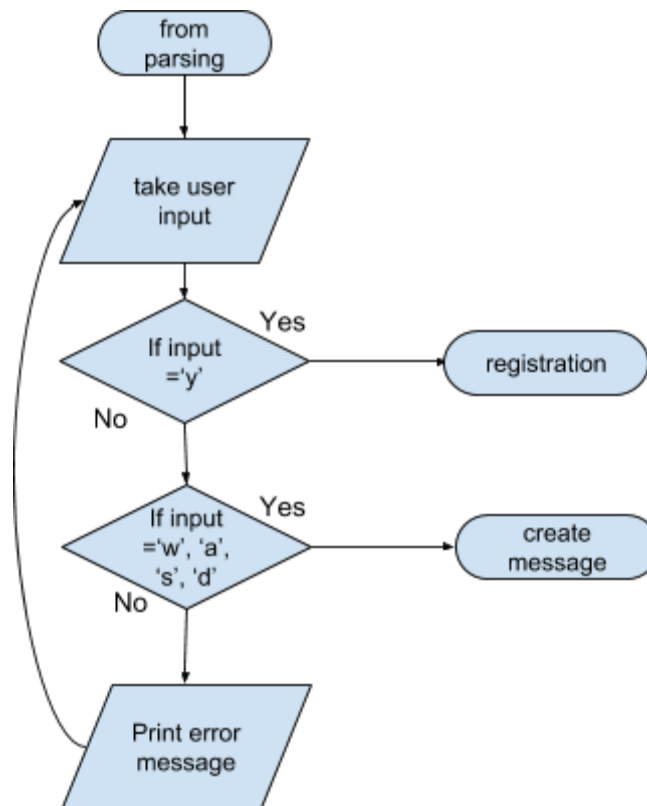
Parsing (server & client)



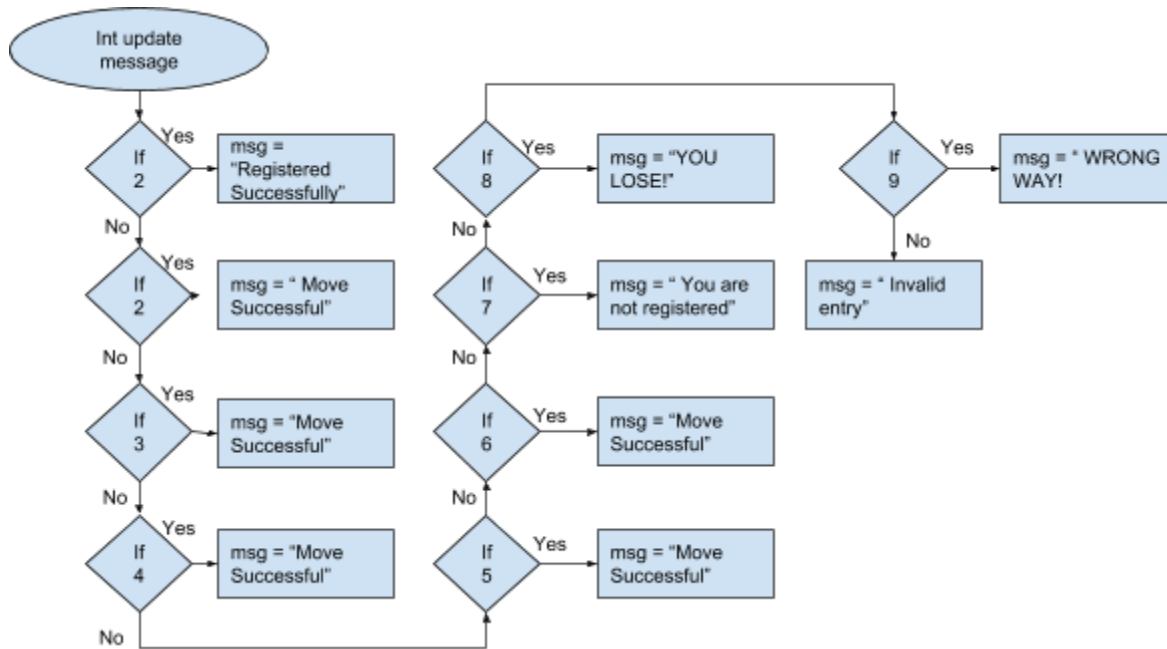
Registration



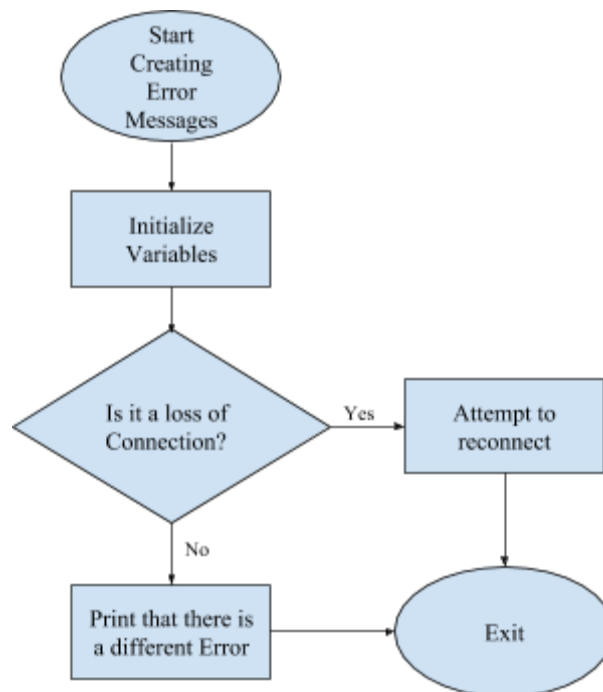
Accepting User Input



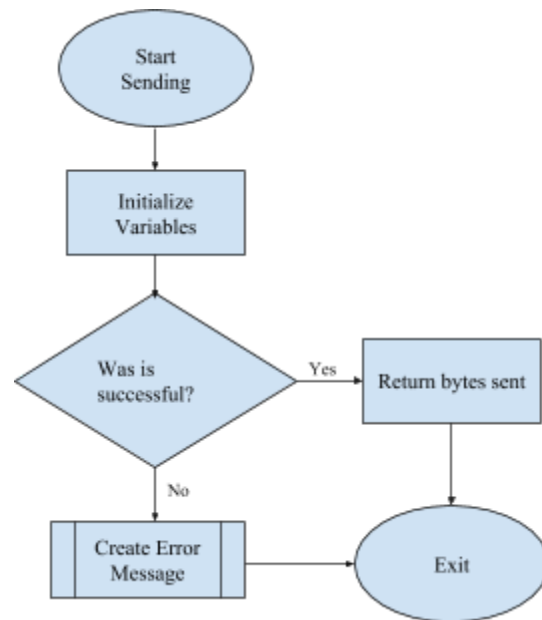
Creating Message



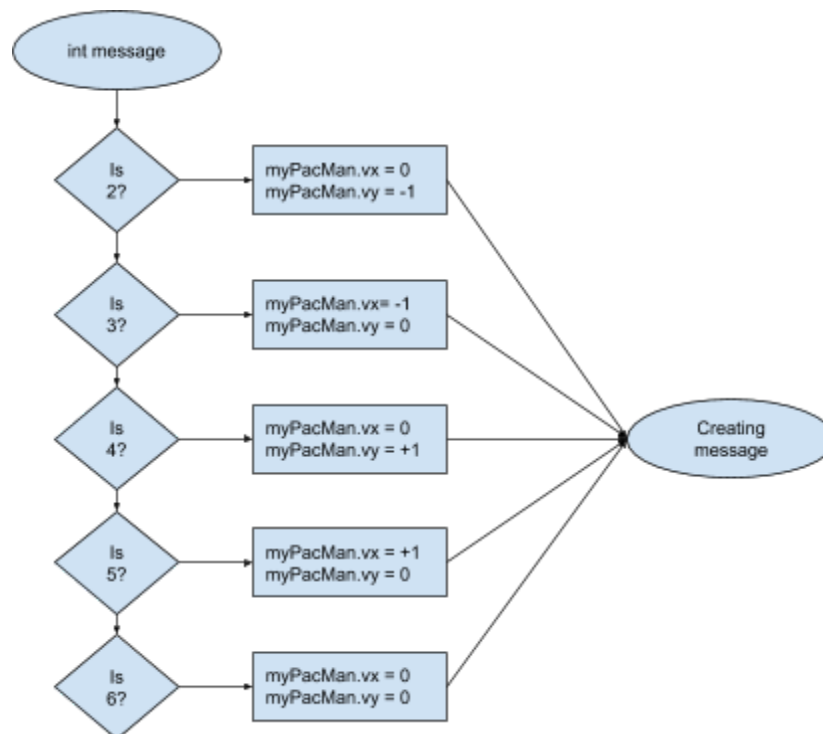
Creating Error Message



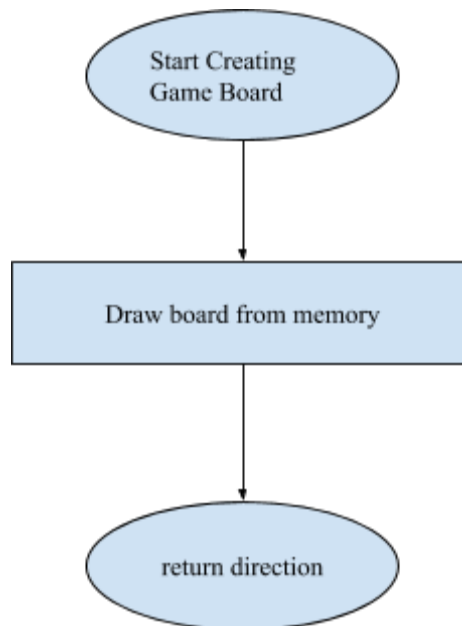
Sending



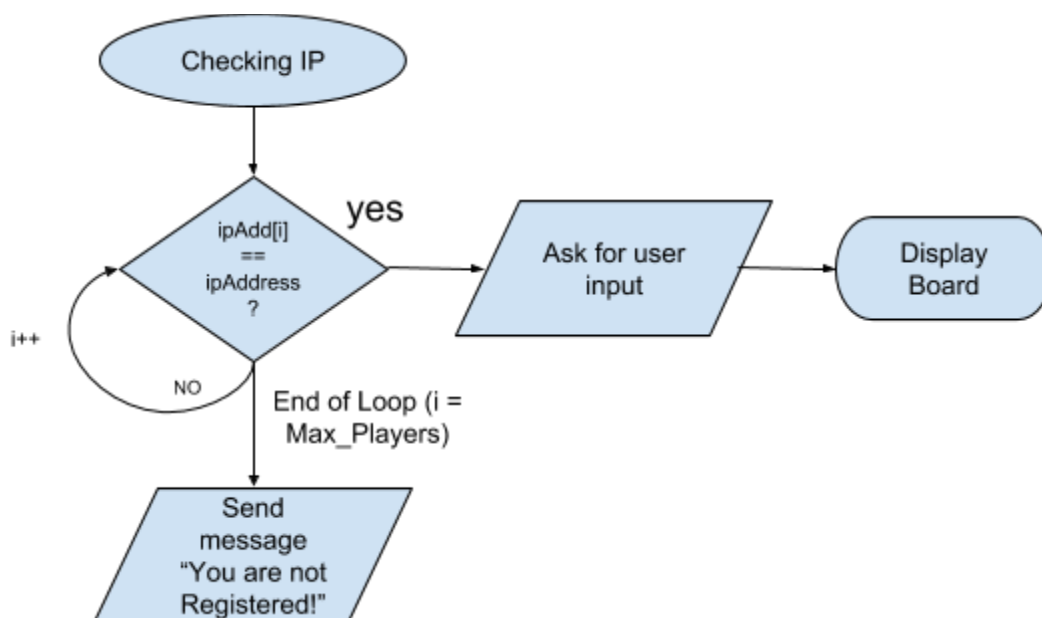
Moving Figures



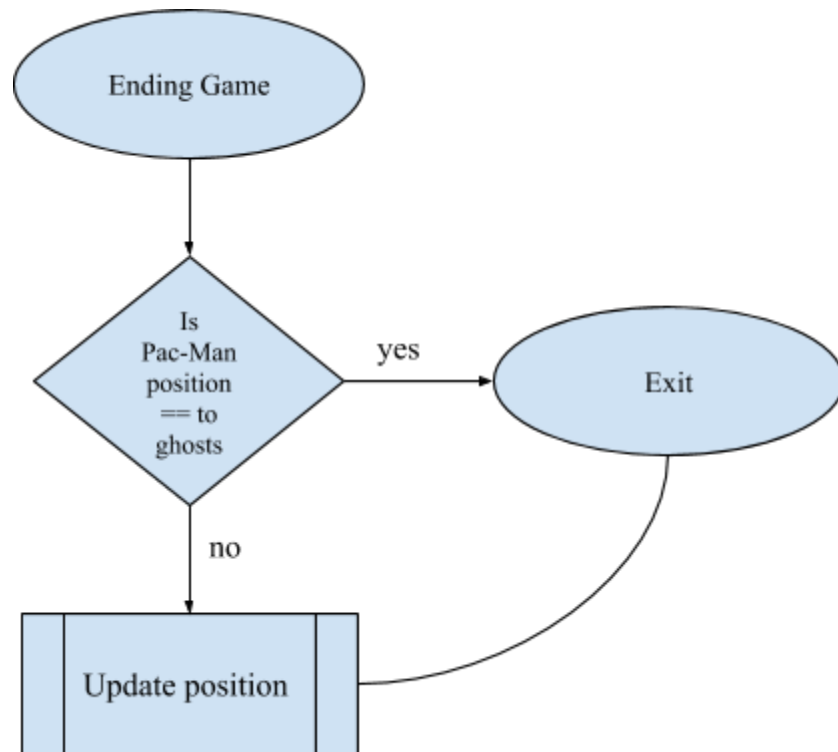
Displaying Board



Checking IP



Ending Game



Implementation

Client

```
#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <ctype.h>

#define BUFFER 1000
#define PORT 2007 //The port on which to listen for incoming data
#define SERVER "10.129.129.13" //ip address of udp server

void sendTo();
void recFrom();

int score = 0;

int main(void)
{
    struct sockaddr_in connectedSocket;

    int s;
    int length=sizeof(connectedSocket);

    char receiveBuffer[BUFFER];
    char userInput[3], *message;

    //clear the buffer by filling null, it might have previously received data
    memset(receiveBuffer,'\0', BUFFER);

    WSADATA wsa;
    //Initialize winsock
    printf("\nInitializing Winsock...\n");
    if (WSAStartup(MAKEWORD(2,2),&wsa) != 0)
    {
        printf("\nFailed. Error Code : %d",WSAGetLastError());
        exit(EXIT_FAILURE);
    }
```

```

}
printf("\n...Initialized.\n\n\n");
//create socket
if ( (s=socket(AF_INET, SOCK_DGRAM, 0)) == SOCKET_ERROR)
{
    printf("\n\nsocket() failed with error code : %d" , WSAGetLastError());
    exit(EXIT_FAILURE);
}
//setup address structure
memset((char *) &connectedSocket, 0, sizeof(connectedSocket));
connectedSocket.sin_family = AF_INET;
connectedSocket.sin_port = htons(PORT);
connectedSocket.sin_addr.S_un.S_addr = inet_addr(SERVER);

printf("Would you like to register to play Pac-Man? (y/n) \n");
while(1)
{
    gets(userInput);
    if (tolower(userInput[0]) == 'y')
    {
        printf("Connecting...\n");
        message[0] = 'y';
    }
    else if (tolower(userInput[0]) == 'n') {
        printf("Closing...\n");
        closesocket(s);
        WSACleanup();
        exit(1);
        break;
    }
    else if((tolower(userInput[0]) == 'w')||(tolower(userInput[0]) == 'a')||(tolower(userInput[0]) == 's')||(tolower(userInput[0]) == 'd'))
    {
        printf("Sending Direction...");
        if (userInput[0] == 'w')
            message[0]='w';
        else if (userInput[0] == 'a')
            message[0]='a';
        else if (userInput[0] == 's')
            message[0]='s';
        else if (userInput[0] == 'd')
            message[0]='d';
    }
}

```

```

else
{
    printf("INVALID INPUT");
}
//userInput[1] = '-';
//message = createMessage(userInput);
//send the message

if (sendto(s, message,sizeof(message) , 0 , (struct sockaddr *) &connectedSocket,
sizeof(connectedSocket)) == SOCKET_ERROR)
{
    printf("\nsendto() failed with error code : %d" , WSAGetLastError());
    exit(EXIT_FAILURE);
}
printf("\nMessage Successfully sent to Server\n");
fflush(stdout);
if (recvfrom(s, receiveBuffer, BUFFER, 0, (struct sockaddr *) &connectedSocket,&length)
== SOCKET_ERROR)
{
    printf("\nrecvfrom() failed with error code : %d" , WSAGetLastError());
    exit(EXIT_FAILURE);
}
printf("Score : %i\n",score);
printf("\nServer Says : %s", receiveBuffer);
printf("\n");
if(strcmp(receiveBuffer,"YOU LOSE!") == 0)
{
    closesocket(s);
    WSACleanup();
    return 0;
}
if (strcmp(receiveBuffer,"Move Successful")== 0)
{
    score++;
}
}

closesocket(s);
WSACleanup();

return 0;

}

```


Server

```
#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
#include <conio.h> // getch()
#include <windows.h>
#include <stdbool.h>
#include <time.h>
#include <ctime>
#include <iostream>
#include <cstdlib>
```

```
#define PORT 2007
#define H 31
#define W 61
#define H2 5
#define W2 51
#define NR_GHOSTS 10
#define BUFFER 10000
#define MAX_PLAYERS 5
```

```
struct coord
{
    int x;
    int y;
};
```

```
struct PacMan {
    struct coord position;
    int vx;
    int vy;
    int lives;
    bool chasing;
```

```

    int food_collected;
};

```

```

struct Ghost {
    struct coord position;
    int vx;
    int vy;
    bool chasing;
};

```

```

struct Ghost allGhosts[NR_GHOSTS];

```

```

struct PacMan myPacMan = {
    {
        .x = 1,
        .y = 1,
    },
    .vx = 0,
    .vy = 0,
    .lives = 3,
    .chasing = false,
    .food_collected = 0
};

```

```

char *msg;

```

```

char heading[H2][W2]=
{
    {"| | ||||| | | | | ||||| |||| |"},
    {"| | | | | | | | | | | | |"},
    {"||||| |||| | | | || | || | |"},
    {"| | | | | | | | | | | | |"},
    {"| | | | | ||||| | | | ||||| |"},
};

```

```

char playfield[H][W] =
{
    { "#####" },

```

```

{ "#          ##          #" },
{ "# #####          ##          ##### #" },
{ "# #####          ##          ##### #" },
{ "#          #" },
{ "# #####          #####          ##### #" },
{ "#          ##          #####          #" },
{ "#####          ##          #####          " },
{ "#####          ##          #####          " },
{ "#####          ##          #####          " },
{ "#####          #####          #####          " },
{ "#####          #####          #####          " },
{ "#####          #####          #####          " },
{ "#          #" },
{ "#          #" },
{ "#####          #####          #####          " },
{ "#####          #####          #####          " },
{ "#####          #####          #####          " },
{ "#####          #####          #####          " },
{ "#####          #####          #####          " },
{ "#####          #####          #####          " },
{ "#####          #####          #####          " },
{ "#          ##          #" },
{ "# #####          ##          #####          #" },
{ "#          #####          #####          #" },
{ "#          #####          #####          #" },
{ "#####          ##          #####          #####          " },
{ "#          ##          ##          #####          #" },
{ "#          #####          #####          #" },
{ "#####          #####          #####          " },
{ "#          #" },
{ "#####          #####          #####          " }
}; // <-- CAUTION! Semicolon necessary!

```

```

void gameTitle(){

```

```

    system("title HAUNTED Server");
}

```

```

void initialize()
{

```

```

    // replace each empty field in the playfield

```

```

// with a food field
for (int i = 0; i < H; i++)
{
    for (int j = 0; j < W; j++)
    {
        if (playfield[i][j]==' ')
            playfield[i][j] = '!';
    }
}

// initialize all ghosts
for (int i = 0; i < NR_GHOSTS; i++)
{
    allGhosts[i].vx = 0;
    allGhosts[i].vy = 0;
    allGhosts[i].chasing = true;
    if (i<5)
        allGhosts[i].position.y = 14;
    allGhosts[0].position.x = 28;
    allGhosts[1].position.x = 29;
    allGhosts[2].position.x = 30;
    allGhosts[3].position.x = 31;
    allGhosts[4].position.x = 32;
    if (i>4)
        allGhosts[i].position.y = 13;
    allGhosts[5].position.x = 28;
    allGhosts[6].position.x = 29;
    allGhosts[7].position.x = 30;
    allGhosts[8].position.x = 31;
    allGhosts[9].position.x = 32;

    playfield[allGhosts[i].position.y][allGhosts[i].position.x] = 'G';

}
} // initialize

void move_ghost()
{

```

```

for (int i = 0; i < NR_GHOSTS; i++)
{
    playfield[allGhosts[i].position.y][allGhosts[i].position.x] = ' ';
    int nx, ny, dir;
    dir = rand() % 4;
    if (dir == 0)
        allGhosts[i].vx = -1;
        //allGhosts[i].vy = 0;
    if (dir == 1)
        allGhosts[i].vy = -1;
        //allGhosts[i].vx = 0;
    if (dir == 2)
        allGhosts[i].vx = +1;
        //allGhosts[i].vy = 0;
    if (dir == 3)
        allGhosts[i].vy = +1;
        //allGhosts[i].vx = 0;

    nx = allGhosts[i].vx + allGhosts[i].position.x;
    ny = allGhosts[i].vy + allGhosts[i].position.y;

    if (playfield[ny][nx] == '#')
    {
        allGhosts[i].vx = 0;
        allGhosts[i].vy = 0;
    }

    allGhosts[i].position.x += allGhosts[i].vx;
    allGhosts[i].position.y += allGhosts[i].vy;

    playfield[allGhosts[i].position.y][allGhosts[i].position.x] = 'G';
}

}

int move_man(int m)
{
    // delete PacMan from old position
    playfield[myPacMan.position.y][myPacMan.position.x] = ' ';

```

```

// compute new desired coordinate (nx,ny)
int nx = myPacMan.vx + myPacMan.position.x;
int ny = myPacMan.vy + myPacMan.position.y;

// test for wall

if (playfield[ny][nx] == '#')
{
    myPacMan.vx = 0;
    myPacMan.vy = 0;
    m= 9;
}

if (playfield[ny][nx] == 'G')
{
    myPacMan.vx = 0;
    myPacMan.vy = 0;
    m = 8;
}

// update PacMan's coordinate
myPacMan.position.x += myPacMan.vx;
myPacMan.position.y += myPacMan.vy;

// is there a food piece at the new location?
if (playfield[ny][nx] == '.')
{
    myPacMan.food_collected++;
    m = 6;
    playfield[ny][nx]= ' ';
}

// put PacMan back again to playfield
playfield[myPacMan.position.y][myPacMan.position.x] = 'M';

return m;
}

```



```

    info.bVisible = FALSE;
    SetConsoleCursorInfo(consoleHandle, &info);
}

void consoleResize()
{
    HANDLE hOut;
    CONSOLE_SCREEN_BUFFER_INFO SBInfo;
    SMALL_RECT DisplayArea = {0, 0, 70, 80};

    hOut = GetStdHandle(STD_OUTPUT_HANDLE);

    GetConsoleScreenBufferInfo(hOut,&SBInfo);

    SetConsoleWindowInfo(hOut,TRUE,&DisplayArea);
}

int parsingMSG(char str)
{
    int msgNumber;
    //char *token = strtok(str, "-");
    char token = str;
    if (token == 'y')
        msgNumber = 1;
    else if (token == 'w')
        msgNumber = 2;
    else if (token == 'a')
        msgNumber = 3;
    else if (token == 's')
        msgNumber = 4;
    else if (token == 'd')
        msgNumber = 5;

    else
        printf("ERROR");

    return msgNumber;
}

```

```

int conv(char *ip)
{
    int num = 0;
    int p1,p2,p3,p4;
    //tok = strtok(ip, ".");
    p1 = std::atoi(strtok(ip, "."));
    p2 = std::atoi(strtok(ip, "."));
    p3 = std::atoi(strtok(ip, "."));
    p4 = std::atoi(strtok(ip, "."));

    num = (p1*1000000000)+(p2*1000000)+(p3*1000)+p4;
    return num;
}

```

```

int* registering(int* ipAddresses, int ipCount, int ip)
{
    //printf("Register\n");
    for (int i=0;i<MAX_PLAYERS;++i)
    {
        if (ipAddresses[ipCount]== ip)
        {

            msg = "Already Registered.";
            break;
        }
    }
    ipAddresses[ipCount]=ip;
    msg = "Use w,a,s or d to move ";
    return ipAddresses;
}

```

```

int checkIP(int message, int* ipAddresses, int ip, int registered)
{
    int saveMessage = message;

    for (int i=0;i<MAX_PLAYERS;++i)
    {
        if (ipAddresses[i]== ip)
        {

```

```

        registered = 1;
        message = saveMessage;
        break;
    }
    else
    {
        message = 7;
    }
}
return message;
}
void createMessages(int message)
{
    switch (message)
    {
        case 1:
            msg = "Use w,a,s or d to move ";
        case 2:
            msg = "Enter move";
            break;
        case 3:
            msg = "Enter move";
            break;
        case 4:
            msg = "Enter move";
            break;
        case 5:
            msg = "Enter move";
            break;
        case 6:
            msg = "Move Successful";
            break;
        case 7:
            msg = "Not Registered";
            break;
        case 8:
            msg = "YOU LOSE!";
            break;
        case 9:

```

```

        msg = "WRONG WAY!";
        break;
    default:
        msg = "Invalid Entry";
        break;
    }
}
int move_figures(int message)
{
    switch (message)
    {
        case 2:
            myPacMan.vy = -1; // cursor up
            myPacMan.vx = 0;
            break;
        case 3:
            myPacMan.vx = -1;
            myPacMan.vy = 0;
            break;
        case 4:
            myPacMan.vy = +1;
            myPacMan.vx = 0;
            break;
        case 5:
            myPacMan.vx = +1;
            myPacMan.vy = 0;
            break;
        case 6:
            myPacMan.vx = 0;
            myPacMan.vy = 0;
            break;
    }
    message = move_man(message);
    move_ghost();
    return message;
}

```

```

int main()
{
    SOCKET s;
    struct sockaddr_in serverSocket, clientSocket;
    char receiveBuffer[BUFFER];
    int clientSocketLength, recv_len;
    int registered = 0; //0 means ip does not exist
    WSADATA wsa;
    int ipCount = 0;
    int ipAddresses[5];
    int message, ip;
    int *addr_ip = ipAddresses;
    clientSocketLength = sizeof(clientSocket) ;

    gameTitle();
    //Initialize winsock
    printf("\nInitializing Winsock...");
    if (WSAStartup(MAKEWORD(2,2),&wsa) != 0)
    {
        printf("Failed. Error Code : %d",WSAGetLastError());
        exit(EXIT_FAILURE);
    }
    printf("Socket Initialized.\n");

    //Create a socket
    if((s = socket(AF_INET , SOCK_DGRAM , 0 )) == INVALID_SOCKET)
    {
        printf("Could not create socket : %d" , WSAGetLastError());
    }
    printf("Socket created.\n");

    //Prepare the sockaddr_in structure
    serverSocket.sin_family = AF_INET;
    serverSocket.sin_addr.s_addr = INADDR_ANY;
    serverSocket.sin_port = htons( PORT );

    //Bind
    if( bind(s ,(struct sockaddr *)&serverSocket , sizeof(serverSocket)) == SOCKET_ERROR)
    {

```

```

    printf("\nBind failed with error code : %d" , WSAGetLastError());
    exit(EXIT_FAILURE);
}
printf("Bind done\n\n");

Sleep(1000);

consoleResize();
initialize();
//keep listening for data

while(1)
{
    hidecursor();
    //system("cls");

    // printf("\n\t\t\tWaiting for data...\n");
    fflush(stdout);
    //show_playfield();

    if((recv_len = recvfrom(s, receiveBuffer, BUFFER, 0, (struct sockaddr *) &clientSocket,
&clientSocketLength)) == SOCKET_ERROR)
    {
        printf("\n\nrecvfrom() failed with error code : %d" , WSAGetLastError());
        //exit(EXIT_FAILURE);
        while(1);
    }

    //print details of the client/peer and the data received
    //printf("\n\nReceived packet from %s:%d\n", inet_ntoa(clientSocket.sin_addr),
ntohs(clientSocket.sin_port));
    //printf("\nClient Says: " );
    //printf("%c", receiveBuffer[0]);
    // Sleep(1000);

    message = parsingMSG(receiveBuffer[0]);
    ip = conv(inet_ntoa(clientSocket.sin_addr));

```

```

if (message == 1)
{
    addr_ip = registering(ipAddresses,ipCount,ip);
    ++ipCount;
    show_playfield();
}
else if ((message == 2)|| (message == 3)|| (message == 4)|| (message == 5))
{
    message = checkIP(message,ipAddresses, ip, registered);
    message = move_figures(message);
    createMessages(message);
    system("cls");
    show_playfield();
}
else
    msg = "INVALID!";

fflush(stdout);

//printf("Final Message: %s\n", msg);

if (sendto(s, msg, 100, 0, (struct sockaddr*) &clientSocket, clientSocketLength) ==
SOCKET_ERROR)
{
    printf("\nsendto() failed with error code : %d" , WSAGetLastError());
    while(1);
}

//else
//printf("\nMessage Sent Back to Client");

if (msg == "YOU LOSE!")
    break;
Sleep( 1000/50);
set_cursor_position(0,0);
}
closesocket(s);
WSACleanup();
return 0;
}

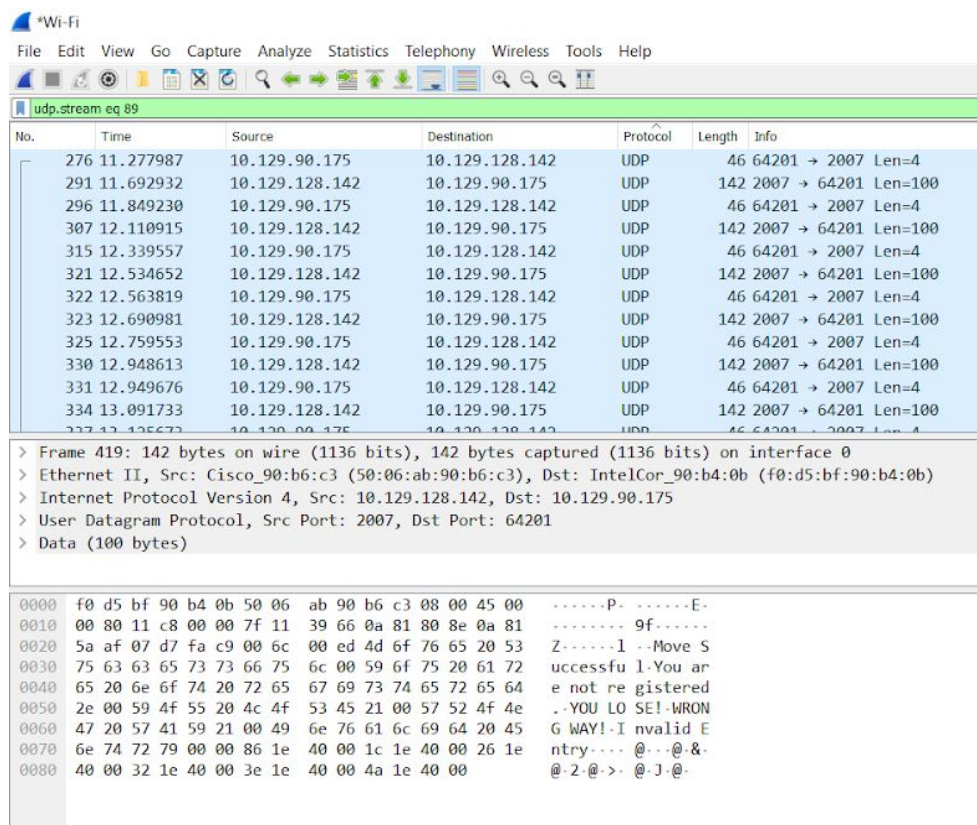
```

Testing

There are two main areas of focus for implementing testing within our program: 1) Network-side and 2) Application-side.

Network:

1. We extensively used Wireshark to ensure our application was achieving a connection between our client/s and server. We needed to be sure that our binded socket set by our server was being connected to by the client/s. Documenting the server and client's IP addresses allowed us to find and follow the ARP requests as well as the UDP thread to see if our messages were being relayed appropriately.

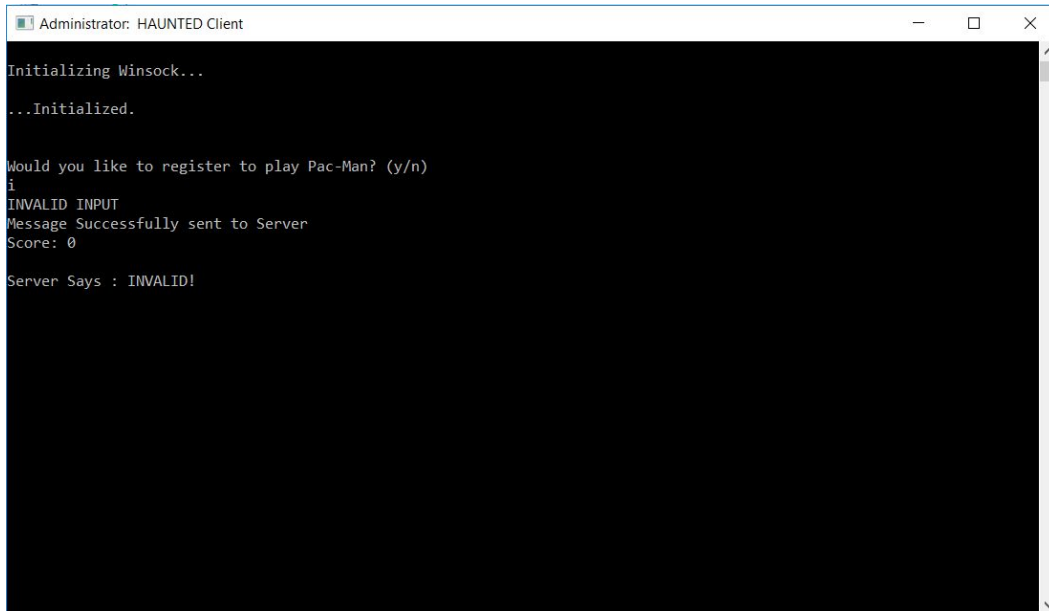


2. In addition, we had to ensure that the firewalls of both the client/s and the server were allowing the data transmission to occur. Attempts to whitelist the connection were made and if those were not met we disabled the firewall temporarily to allow for the connection to be made.

Application:

Client-Side:

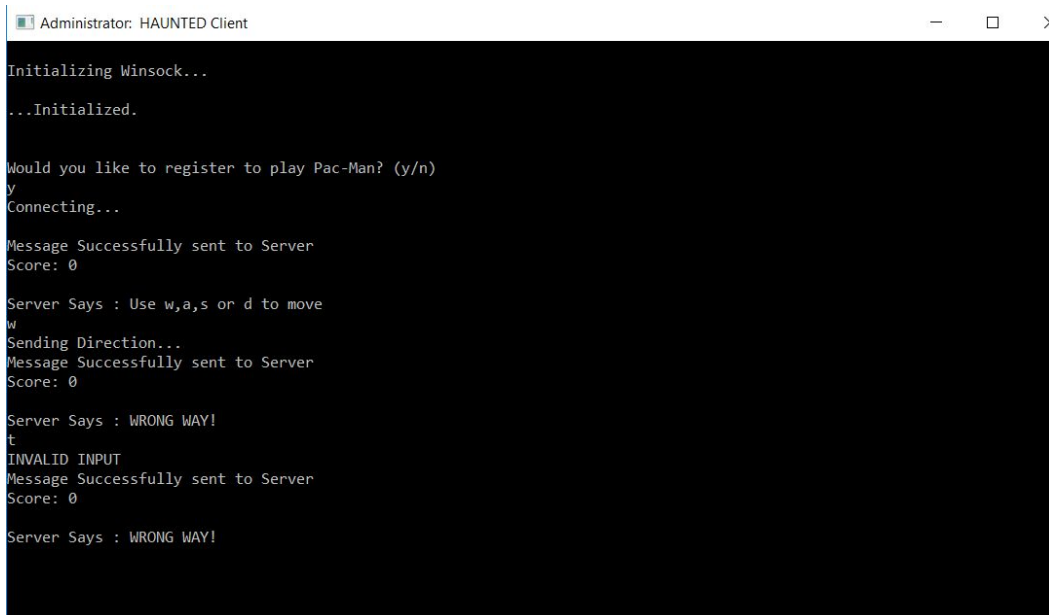
1. Registration needs to make sure that it only accepts a 'y' or 'n' when asked about the status of this. All other inputs needs to give an error and prompt for an appropriate response.



```
Administrator: HAUNTED Client
Initializing Winsock...
...Initialized.

Would you like to register to play Pac-Man? (y/n)
i
INVALID INPUT
Message Successfully sent to Server
Score: 0
Server Says : INVALID!
```

2. Movement state needs to ensure that only 'w', 'a', 's', 'd' are accepted and sent to the server. Any other input needs to return an error and prompt for an appropriate input.

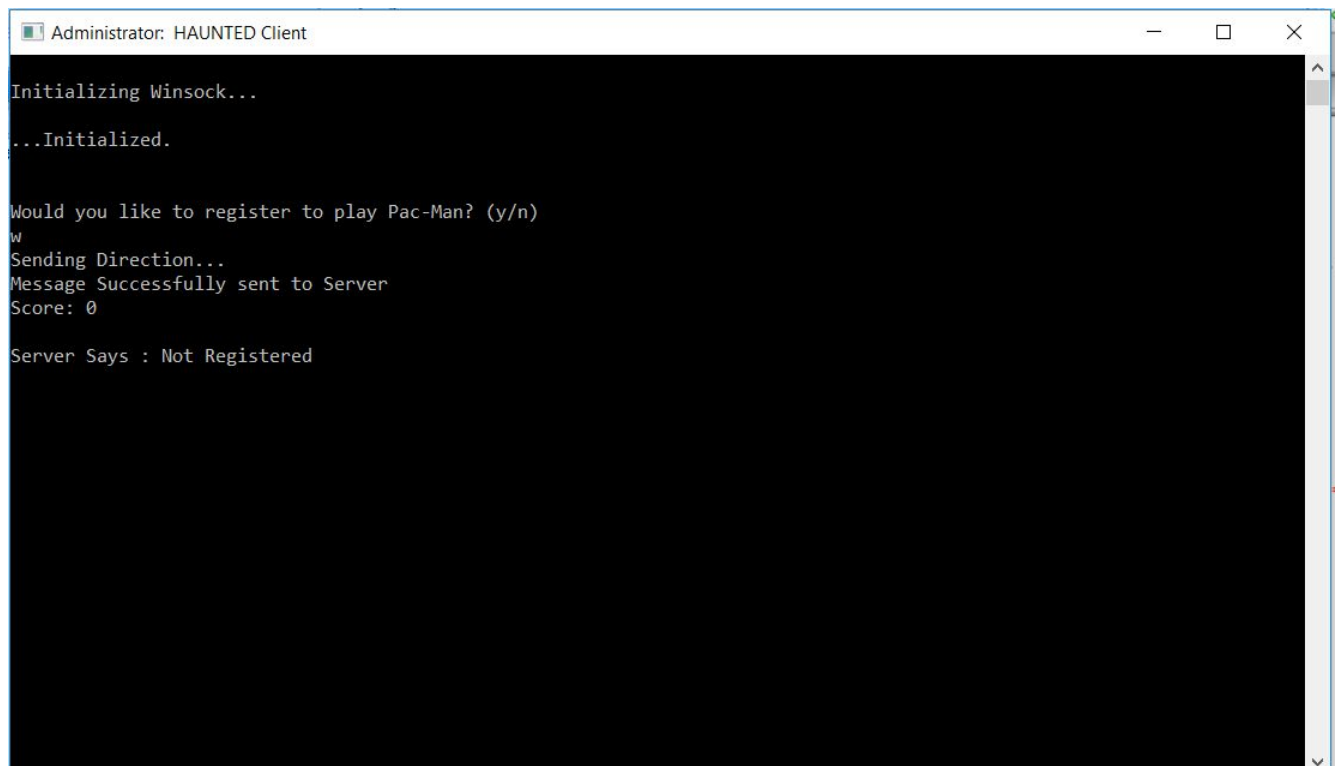
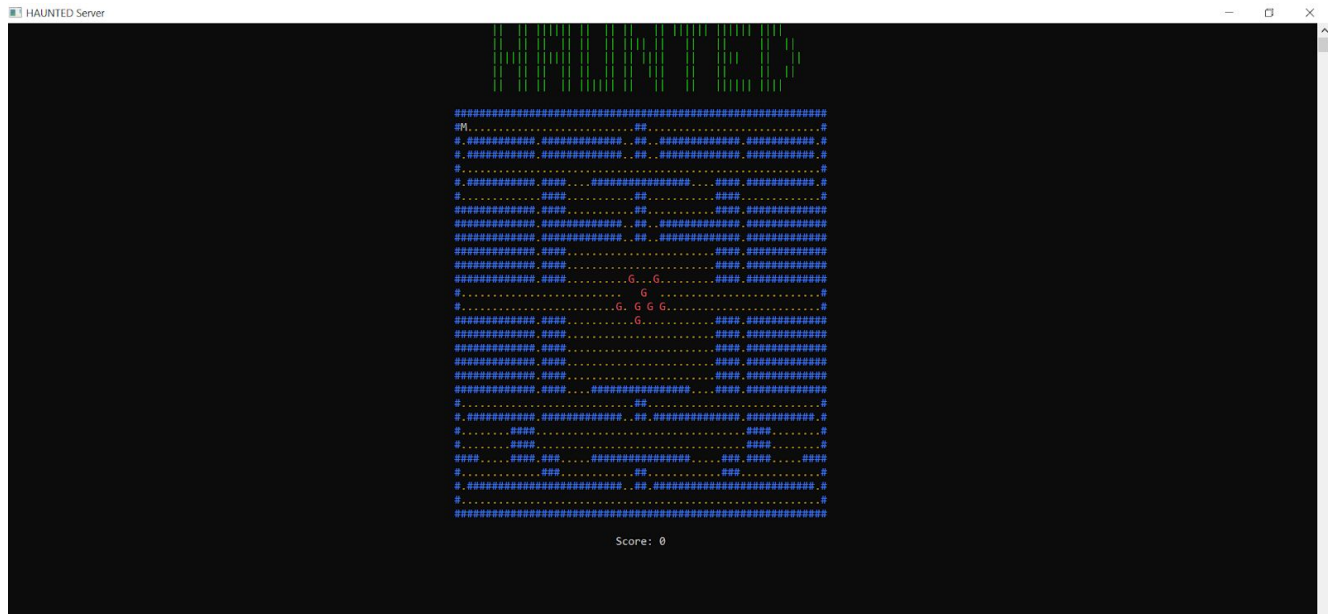


```
Administrator: HAUNTED Client
Initializing Winsock...
...Initialized.

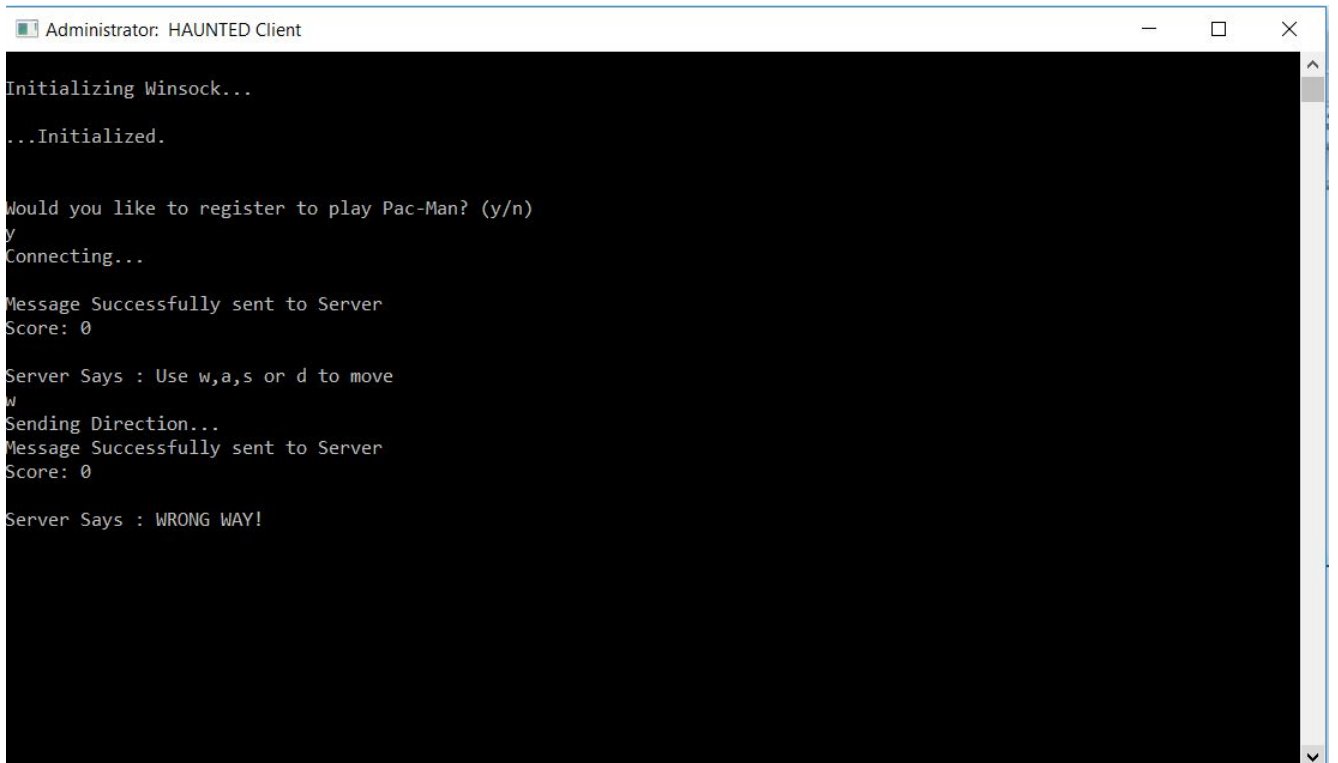
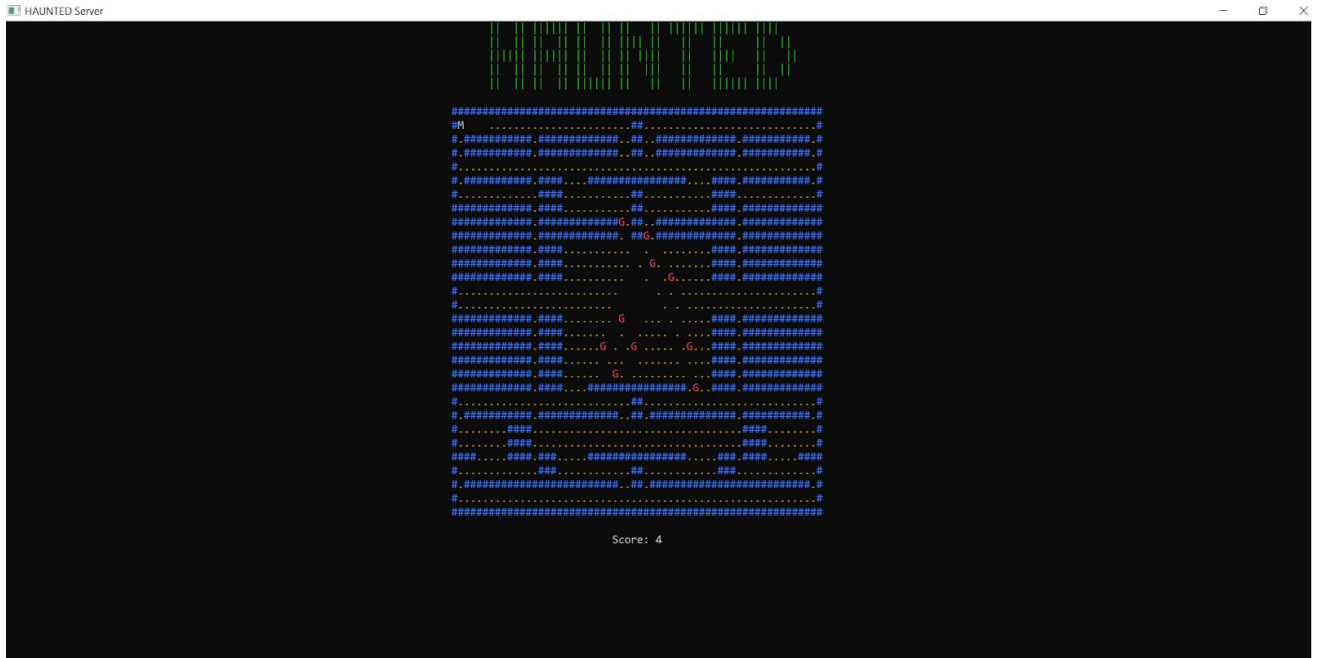
Would you like to register to play Pac-Man? (y/n)
y
Connecting...
Message Successfully sent to Server
Score: 0
Server Says : Use w,a,s or d to move
w
Sending Direction...
Message Successfully sent to Server
Score: 0
Server Says : WRONG WAY!
t
INVALID INPUT
Message Successfully sent to Server
Score: 0
Server Says : WRONG WAY!
```

Server-Side:

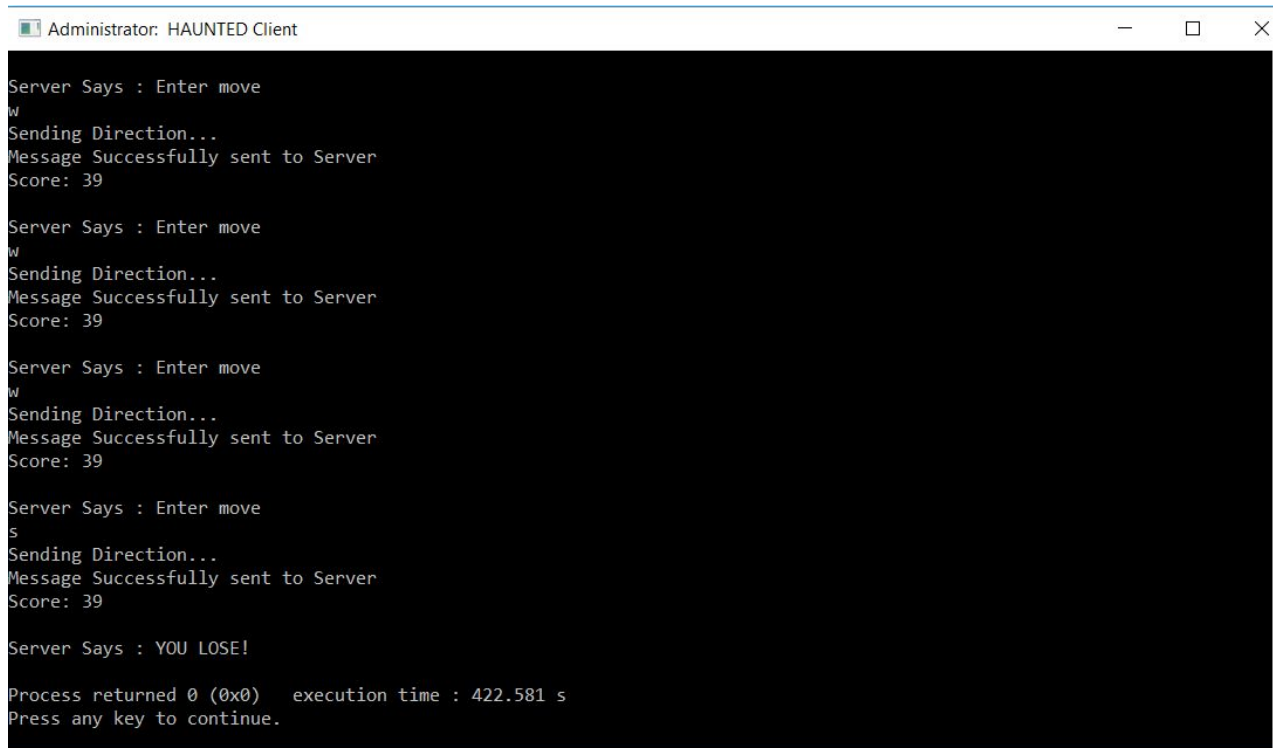
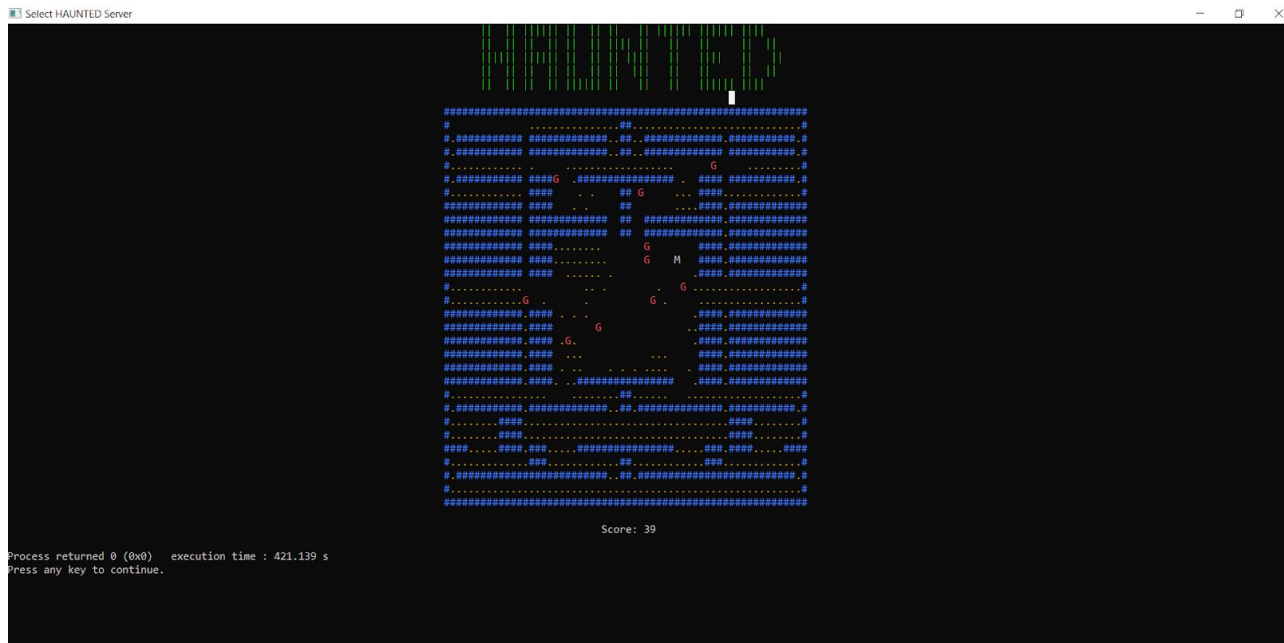
1. Test for how the server and client responds when given a movement direction during registration phase.



2. Test for how the server and client responds when given a movement direction towards a wall.



- Test for how the server and client responds when the game ends (pac man runs into a ghost).



Documentation

“Haunted” is a multiplayer game that makes use of multiple clients and a single server to play a game very similar to the traditional game of Pac-Man. Clients are able to join the game at any time during game play but requires one player to register to start the game. Only clients who are registered are eligible to play. After registering, the client will use the ‘w’, ‘a’, ‘s’ and ‘d’ keys to move the “man” throughout the playfield. The keys represent the direction in which the “man” will move whether it is ‘up’, ‘left’, ‘down’ and ‘right’, respectively. The aim of the game is to avoid the ghosts and collect points by eating the dots on the screen. All players are controlling the single “man” that is seen on the server screen. If the client makes a move and eats a dot, it will add to both his/her personal score and the overall game score. If the “man” collides with the “ghost” the game will end for all players.

Generalization

“Haunted” could have been broadened to closer mimic a traditional Pac-Man Game by implementing actually graphics and implementing the larger dots that would provide the capability of the player eating the ghost. Another generalization could involve having each client seeing the game on their screen and each time a new player enters the game his character will appear as a ghost on other the other client’s screen. This would have each game on the client’s screen appear as a traditional Pac-Man game where the server will keep track of all the game data for all clients.