CMPE 207 Network Programming and Applications
Fall 2014

MULTICLIENT ONLINE TIC-TAC-TOE

Project Report by:
Hrishikesh More

Other Team Members:
Divya Kamat
Mradula Nayak
Shanu Jain

Course Instructor
Dr. Wei Xu
wei.xu@sjsu.edu
May, 2015

Table of Contents

1. ABSTRACT

Online gaming is becoming very popular now a day. In these online gaming systems there is a central server which monitors over the game. There is multiple clients connected to this server to play game with each other. These kind of server are distributed Client – Server model. This document gives a brief overview and implementation of multicleint online gaming of Tic-tac-toe. This document covers the basic server client messages being exchanged during the game.

2. INTRODUCTION

Online gaming is gripping world's attention. This online gaming is based on distributed server client model. According to this server client model there are three main parameters to be kept in mind i.e. concurrency of components, lack of a global clock, and independent failure of components.

i) Concurrency of clients:

The components taking part in the gaming here multiple machines should achieve concurrency. They should work independently and at the same time simultaneously. In multiple clients gaming concurrency is important because slight latency or delay will produce lag in gaming. So it is important that all clients participating in the game should be concurrently running and has the feel of real time gaming.

ii) Lack of global clock:

As multiple machines are working simultaneously these machines should also have independent clocks. Independent clock signifies that these machines are not dependent on other machines and have their own clock cycles for programming.

iii) Independent failure of components:

The online gaming should be such that it should not fail if there is a failure in any component. The game should run also if the main server fails. This can be achieved by implementing a backup server. This server will be up as soon as the main server fails. All the information is transferred to this server and the game goes on.

Following are the information stored by a online game server:

- i) TCP/IP configuration of all clients connected.
- ii) IP address of all the clients connected.
- iii) Available clients for game
- iv) Running game status

Following are the information being exchanged between server and client:

- i) Number of clients connected to the server.
- ii) Status of the game.
- iii) Moves for the game.
- iv) Selecting the payer to start a new game.

3. PURPOSE

The purpose of this project is to simulate an online gaming server-client interaction environment that is similar to a production version of online gaming software deployed on commercial routers.

The software framework follows the same protocol messages to assign an ID to client requesting to join a game and other services like initiating a game, maintaining status of game etc. It would also provide additional information based on message types requested by client. The software works with only IPv4 clients. In future this can be enhanced to support IPv6 clients.

4. MOTIVATION

Since the advancement on in broadband communication and high speed internet there have been many applications developed. One of such applications is online gaming. This technology has been seeing an exponential rise in the number of devices connecting to it. The sense of real time multiplayer gaming has been very eye catching and addictive.

The curiosity to understand and mimic how the multiple client online software works in consumer and commercial platform is the motivation behind our project. Our project follows all the standard protocols of a commercial distributed server client model and provides the same functionality.

5. Application Protocol:

0xA5	0x010A	9	DATA	0x5A
Header	Message ID	Message Length		EOM

Above shown is an application protocol we designed. There is an exchange of messages in server and client regarding the game. These messages carry data essential required for playing the game. These messages are identified as game related by looking at the header. The header defines whether the given packet is a game related or not. 0xA5 denotes that the given packet is a game related packet. After knowing the packet is a game related packet we need to distinguish between the messages contained in them. There are 23 message types varying from registering a new user, starting a new game, score of the game etc. Thus the type of the message is later defined by the message ID field. According to these message ID's there are various actions performed.

The length of the message may vary from 0 byte to 9 byte depending upon the length of DATA. Therefore message length field is introduced to accommodate variable length of data. Thus looking at the length the amount of data can be read reducing the processing time. Further the Data is passed which can vary from 0 bytes to 9 bytes. This data can be the clients ID, status of a game etc. The Data segment in frame is read and necessary output is shown on the client side and necessary actions are taken on the server side. Further to determine the end of the message there is End Of Message field, which denotes that the message has ended.

Following are the server and clients messages:

5.1 Server Messages Structure:

Message Type	Message ID	Message Length	Data[0:7]
Welcome Message to	0x0100	0	0

Client			
User Registration Successful with user ID XX (XX is the user ID assigned to client by the server)	0x0101	1	Data[0] = XX
User Registration Unsuccessful	0x0102	0	0
Request Player for a game with user XX. (XX is the user who initiates the game)	0x0103	1	Data[0] = XX
Game Session Start Success	0x0104	0	0
Game Session initiation failed	0x0105	0	0
Game Won	0x0106	0	0
Game Lost	0x0107	0	0
Game Tie	0x0108	0	0
Display Stats to Client	0x0109	8	Data[0] = Number of games currently in session Data[1] = Top 1 score of all time Data[2] = Top 2

			score of all time Data[3] = Top 3 score of all time Data[4] = Top 4 score of all time Data[5] = Top 5 score of all time Data[6] = Client maximum score Data[7] = ?

5.2 Client Message structure:

Message Type	Message ID	Message Length	Data[0:7]
reguser - register user	0x1000	0	0
newgame XX - Start a new game with player XX	0x1001	1	Data[0] = XX
endgame - End a game in session	0x1002	0	0
select RXCY	0x1003	2	Data[0] = X Data[1] = Y
viewstats - View Game Statistics	0x1004	0	0
Yes - Accept Server Request	0x1005	0	0

No - reject Server Request	0x1006	0	0

6. Distributed Server-Client Design:

The server client design we have used is a distributed server client. The following the diagram of a server client model:

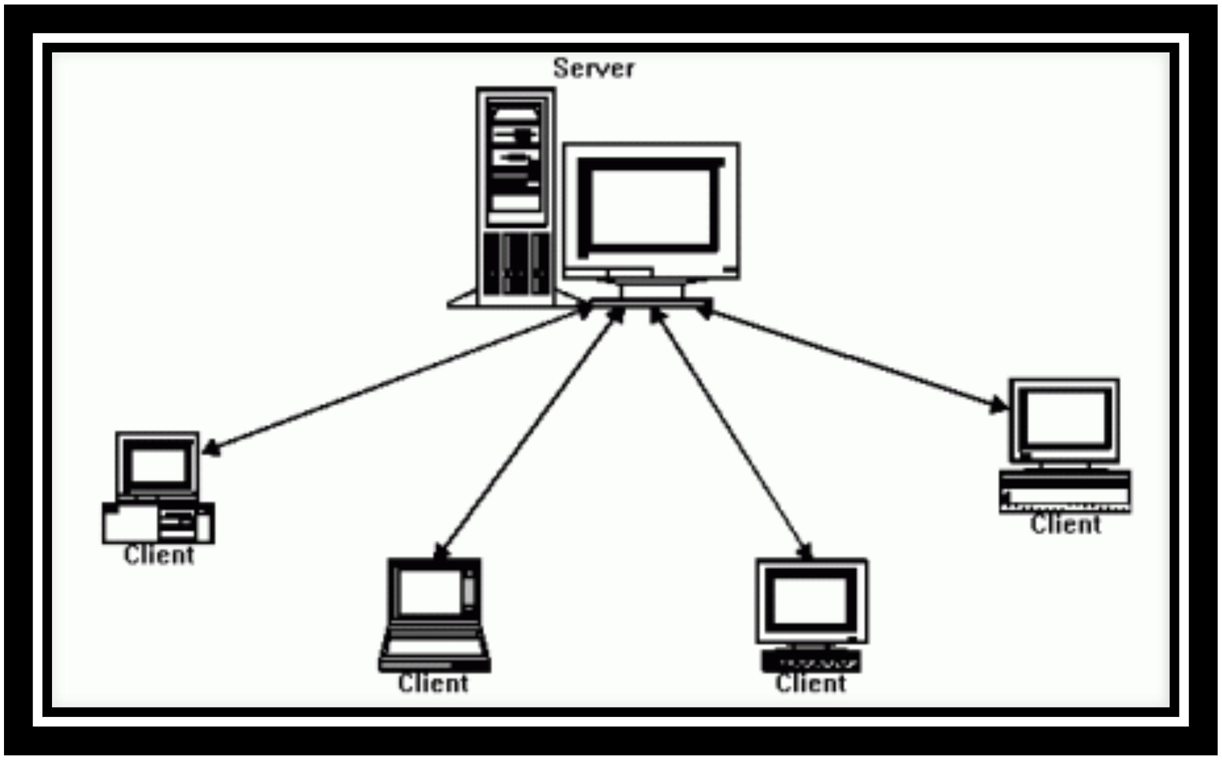


Figure 1. Distributed Server Client

As shown in the diagram there is one server with many clients connected to it. One of the main functionality aimed in this project is to provide a centralized location for sharing the game related data between the clients. This can be achieved by providing a centralized sharing location. This sharing location can be a server. Clients can connect to this server and then they would try to access the data as well as send the data to be stored on the server. This data could be the clients ID, the clients' game status, moves etc. So the centralized shared location would be on server and clients can connect to that server with proper authentication inputs. Hence there is a need of server-client design.

7. Project Modules:

7.1 Socket Framework:

The socket framework for the online gaming is shown as follows:

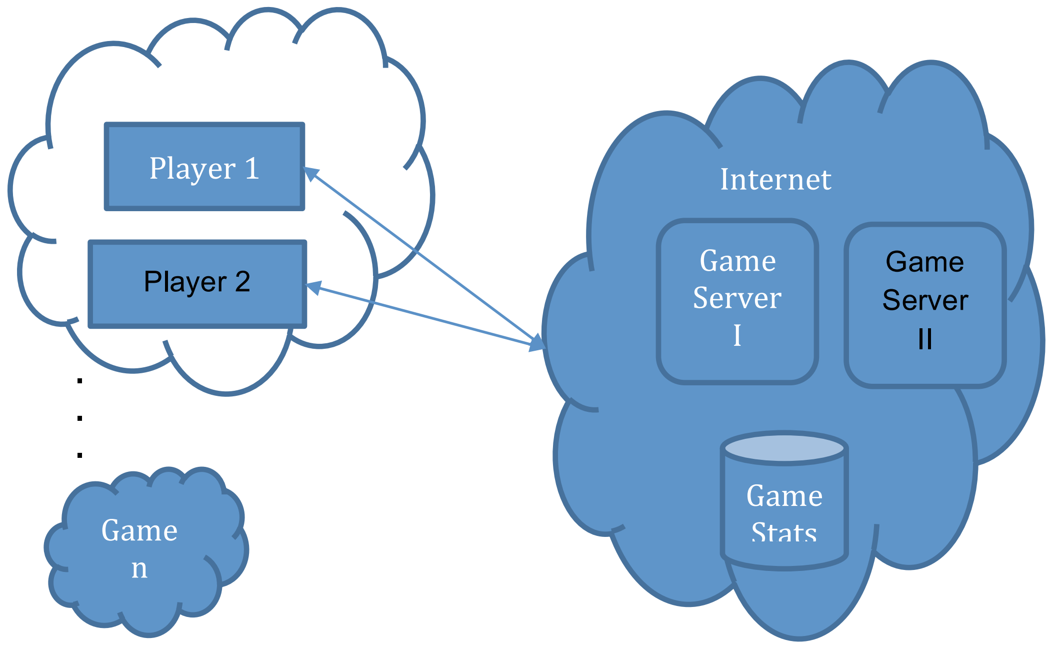
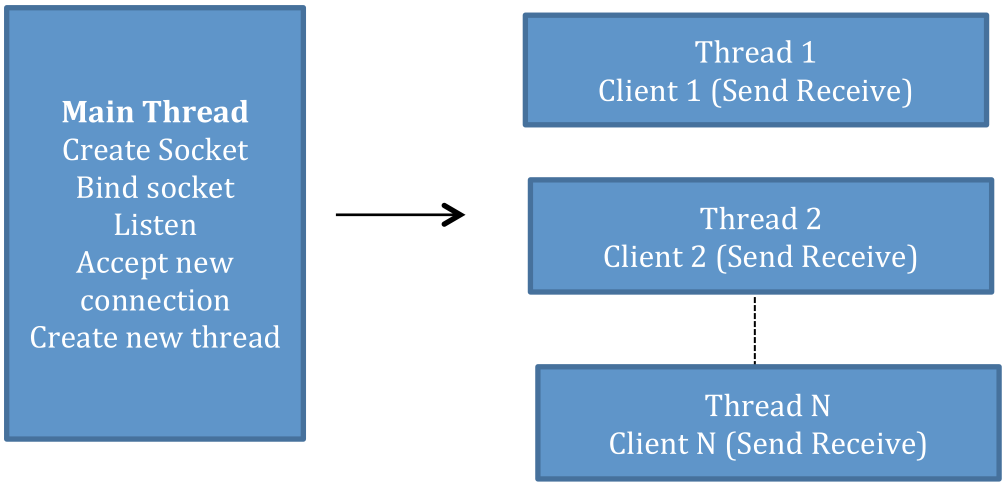


Figure 2. Distributed Server Client Socket Framework

We are using **TCP connection** between server and client. TCP provides reliable, connection-oriented, byte stream service. The processes must establish connection before transmitting the data. In our design, we need the receiving end to get ordered data packets so that header and EOM can be recognized. Hence we opt for TCP socket programming.
The server is a multithreaded server which creates a new thread for each new client. The socket framework for this is as follows:



7.2 Server Design:

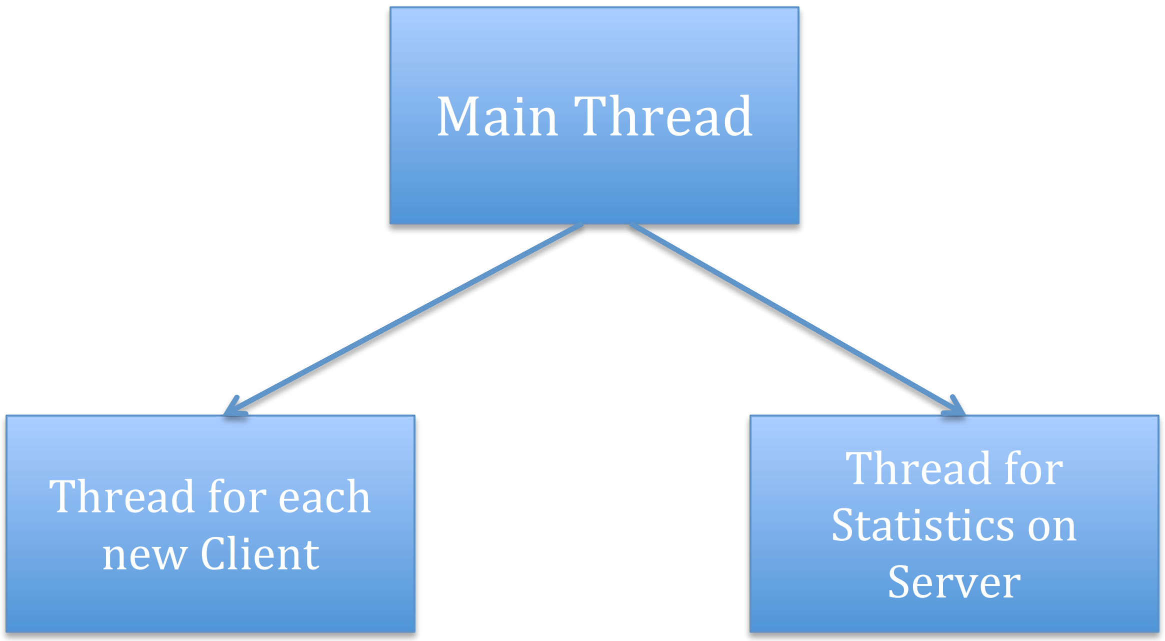


Figure 3. Server Design

The server is divided in three parts main tread, thread for each client, thread for statistics on server. The main thread, as we know is the parent thread, which creates the thread for each, connecting new client and thread for statistics depending upon the request being serviced.

The main thread executes the following functions:

1. Create a TCP socket for server.
2. Bind the IP and port number to the socket.
3. Listen to the port for client requests
4. Accept client requests and create a new thread.
5. Create a new thread to display the statistics of the game.
6. Create a mutex lock for the vector of registered users.
7. Create a mutex lock for the vector for running games.
8. Create a mutex lock for vector of the command queue.

The thread for new client executes the following functions:

1. Receive the message from the client and parse the message ID.
2. Based on the value of message ID, execute the corresponding function.
3. There are 4 functions called by the client thread:
    - handleRegUser() – When message ID received is 0x1000
    - handleNewGame() – When message ID received is 0x1001

- handleEndGame() – When message ID received is 0x1002
- makeAMove() - When message ID received is 0x1003

There four crucial functions of the thread for new client. The main thread when encounters to register a new client it creates a thread for that client. This thread registers the new user and specified an ID specific to that user. After successful registration the client can view the number of clients connected and request to play with any of them. When the client requests a new game with another client the server checks with the other client and initiates a new game with it. This new game is also a separate thread with its own game ID. The clients can now send messages and play. When the game gets over the game thread is deleted and the two players will be available to play again.

The other thread created by the server is thread for statistics. This thread acts as a database. It maintains all the logs such as current connections, completed connections, average completion time. When the games are initiated it maintains the number of active games and total registered users. A structure for statistics is defined in the server which holds the following data:

- Current connections
- Total number of registered users
- Total number of running games
- Completed connections
- Average completion time.

Three threads are running continuously to print the above values.

1. Thread which updates the current connections, every time a new client request is accepted.
2. Thread which decrements the current connections when a client ends.
3. Thread which displays the statistics on the server continuously.

7.3 Client Design:

Unlike the sever client is a single process with no multi threading involved. There are certain important functions that the client needs to perform. Every client needs to be connected to server efficiently. The connection we have used to make a reliable connection is TCP. As in TCP there are no loss of packets and it being

connection oriented will have less error. There is an exchange of packets between server and client. Depending upon
different Message ID's different messages are identified and the client takes appropriate actions. The different messages exchanged are given as follows:

- GAME_STARTED
- GAME_START_FAIL
- MAKE_A_MOVE
- MOVE_SUCCESS
- GAME_WON
- GAME_LOST
- GAME_TIE
- ERROR_MSG

After the user is registered on the server the client have a few information request like finding the number of clients connected on server. After the information exchange the client can initiate a game with other connected clients. After the successfully starting a game the client enters a loop, which does not end until the game, ends. During this loop the gaming algorithm is executed to find the game win or loss. The client is a single process application. Each client executes following function:

· Create a socket for the client.
· Connect to the server socket
· It also displays the registered user list on client screen.
· Send command to register the user, start a new game with another client.
· Make a move by selecting the row and column.
· On every MOVE_SUCCESS, the tic-tac-toe game is displayed on the client screen.
· GAME WON, GAME LOSE, GAME TIE cases are also displayed on the client screen.

7.4 Game Algorithm:

Every game needs a game algorithm to maintain the status of the game and deicide whether the game results in win, lose or tie. There are two characters in the game '*'

and ‘o’. When there are three consecutive either ‘*’ or ‘o’ then the respective client wins and the other client losses. If there are no three consecutive ‘*’
or ‘o’ then the game has resulted in a tie. There is a structure, which maintains the status of the ongoing game.

0	*	0
*	0	*
*	0	*

Figure . Game structure

Every Game has a structure which stores:

player1_index; // Socket ID of 1st player

player2_index; // Socket ID of 2nd Player

GameArray[9]; //0 for empty,'*'or 'o'

player1_status; // win, lose

player2_status;// win, lose

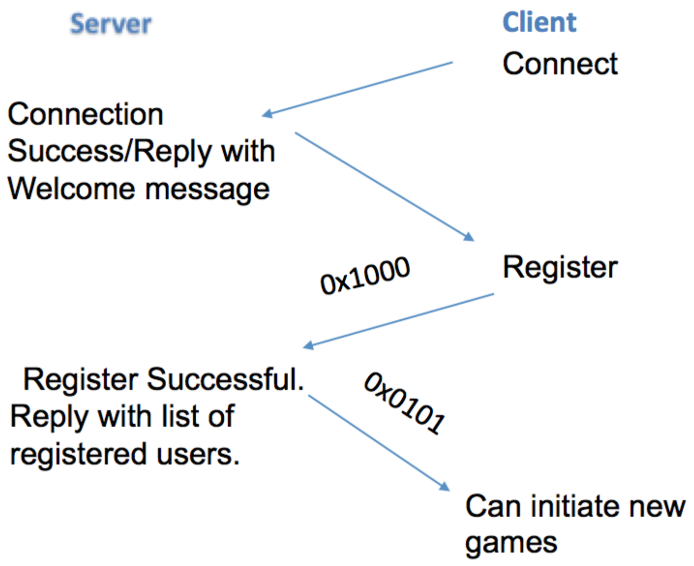

Server
Client
Connect
Connection Success/Reply with Welcome message
0x1000
Register
Register Successful. Reply with list of registered users.
0x0101
Can initiate new games

Figure . Connecting the client

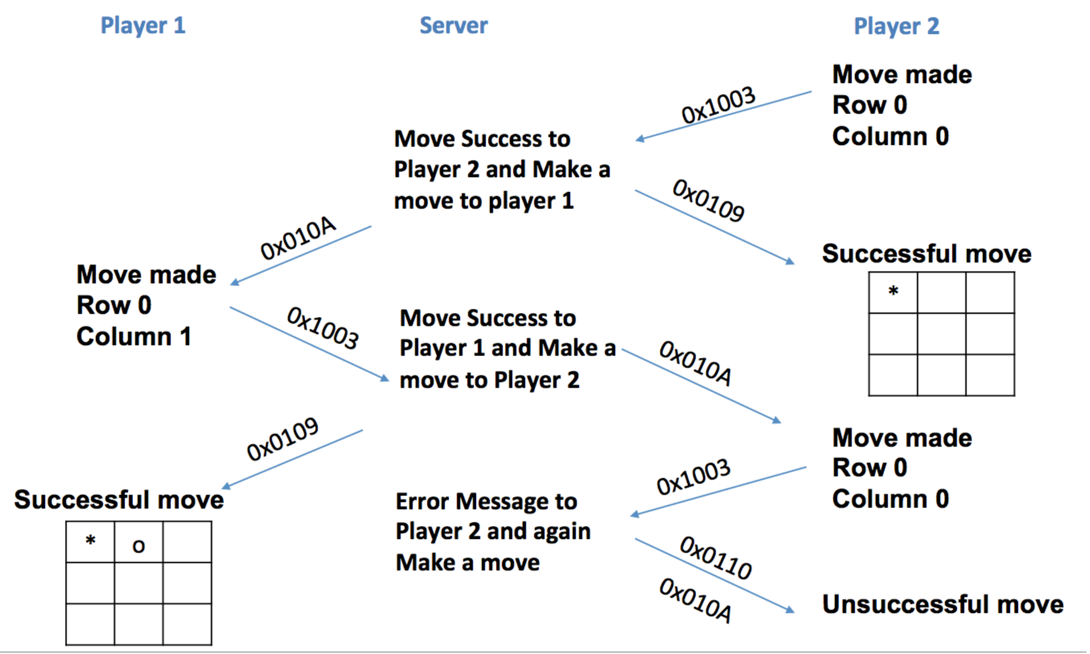


Figure . Gaming Algorithm

8. Error Handling:

8.1 Error handling in Clients:

1. The user shall not be able to select an opponent who is not in the list of registered users. An error message will be displayed to the user.
2. The user shall be not allowed to make an invalid move.
3. The user shall not be able to make more than one move at a time.
4. The user shall not be able to use the opponent's symbol.

8.2 Error Handling in Server

1. The user shall be notified if one of the user disconnects in between.
2. If the user does not move within a limited time, a warning/timeout message shall appear.

9. Testing Plan:

The following test cases will be checked during testing phase.

9.1 Client Testing

The client module is tested for several functions:

- Connection establishment and disconnection.
- The client process should end when the window is closed.
- To test the start of a new game after a disconnection.
- Select opponents from the list of active registered users. If there are no active users currently, the client cannot proceed the game.
- To test if choices of * and 0 given by the user are displayed correctly on the corresponding row and column. If the row/column no. entered is besides 1-3, an error message is displayed.
- To test if the client module works on a single host for several users.

9.2 Server Testing

- To test the algorithm of the game written on the server.
- To check the connection and disconnection request handling on the server.
- To check the validity of the database of connected users.
- Since the server is multithreaded, we need to ensure that no data corruption will take place when multiple games are being played at the same time. Each game will have its own data storing mechanism to prevent any exchange of results.
- The test case would also include a sudden crash to check the transfer of services to another server in case of failure.
- The validity of statistics of each game is tested by cross verifying the results noted manually.

9.3 Integration Testing

When the server and client modules are connected together, we test the integrated network performance.

- The socket programming of the connected system is tested within the network.
- Test if the client and server communication over the LAN is working.
- The speed and concurrency of the network with multiple clients and server running at the same time

10. Performance Evaluation:

In our programming we have chosen multithreading over multiprocessing. The threading module uses threads, the multiprocessing uses processes. The difference is that threads run in the same memory space, while processes have separate memory. This makes it a bit harder to share objects between processes with multiprocessing. Since threads use the same memory, precautions have to be taken or two threads will write to the same memory at the same time. This is what the global interpreter lock is for.

To solve the problem of shared memory we have used mutex locks in our program. Mutex stands for mutually exclusive. The following is the code used to implement mutex lock and unlock mechanism.

Initializing a Mutex Lock:

```
 //Initialize mutex for list of registered users
 pthread_mutex_init(&regUsersList_mutex, 0);
// Initialize mutex for list of running games
pthread_mutex_init(&runningGameList_mutex, 0);
// Initialize mutex of queue
pthread_mutex_init(&regUserQueue_mutex, 0);
```

Mutex lock:

```
(void) pthread_mutex_lock(&regUserQueue_mutex);
 if(regUserListIndex != -1)
{
queueString = regUserQueue.at(regUserListIndex);
}
(void) pthread_mutex_unlock(&regUserQueue_mutex);
```

Using Mutex lock when a particular thread is accessing the data then the other thread wont be able to access that data. It will wait in a queue for the Mutex to be unlocked.

11. SCALABILITY:

The server that we created can have about accept upto 100 clients at a time. There can be an ad-hoc network created and the server machine can be assigned IP statically. Similarly when the client program is initiated on another machine it should be connected to that ad-hoc network and the server address should be specified so that it gets connected to the server. We can thus play our game on multiple machines. We can make minor modifications in the program and the server can extend upto multiple clients as preferred.

12. ENVIRONMENT AND TOOLS USED

Environment : MAC OS X

Software Tools : Text-Wrangler.

Presentation Tools : Prezi, Screen capture software, Windows Powerpoint.