Objective:

Write a simple remote execution system. This requires putting into practice what has been taught about systems programming.

Introduction:

In this assignment you need to implement two programs: one that runs on a *Game Server* and is the game manager, and one that runs on a *Game Client* and is used by players to communicate with the Game Server throughout the game.

The game you will be implementing called “Numbers” (a description of the game appears bellow).

The Game Client and Server are two independent programs that run on different machines at the same time and communicate through a network. (The programs can also be run on the same machine for testing).

There will be an open port on the Game Server that will listen for join requests. To allow each player (client) to join the game, a separate communication channel will be setup.

The Protocol:

For the communication between the players and server to work, both sides need to agree on some communication rules, also known as a “protocol”.

Every game is divided into 4 stages:

1. **Initiate the Game Server** - The port number, the name of the game and other arguments to set up the game are given to the server as a command line arguments.
2. **Join** - The server awaits for players (clients) to join. When a new player joins, the server sends them a text message saying “welcome to the game”. The server moves to the next stage only after the required number of players has joined.
3. **Play** -The server iterates amongst the players (in their joining order) and sends/receives messages based on the game (details below). During this stage, players may leave the game, and if the number of players falls below the needed minimum, the game ends. No new players can join at this stage.
4. **Game Over and Announcing Results** -All players receive notification messages from the server announcing the end of the game and whether they won or lost. All resources (sockets) should be stopped and closed after the game is finished.

During the *Joining stage*, the server opens a socket connection based on the port given in the command line. When using a socket connection, a joining request is just a client program querying the remote server program that is listening on the port. If the connection query is successful, the server sends the message "welcome to the game" to the client.

During the *Play stage*, messages are being sent between the server and the client using the socket connection.

There are 4 types of messages the server may send to clients:

1. **`TEXT`** ***<text>*** - A client receives this message needs to print the contents of *<text>* to the screen as is. For example, if the client receives the message:

   ```
   TEXT It is your tern
   ```

   The client should print to the screen in a new line the string:

   ```
   It is your tern
   ```

2. **`GO`** - A client receives this message should wait for a user input while the server waits to receiving the input from the client.
3. **`END`** - This message indicates the game ended. A client receives this message should close the connection and stop running.
4. **`ERROR`** - This message indicates an error.

And there are 2 types of messages a client may send to the server:

1. **`MOVE`** ***<move>*** - Informs the server on a game move (This is a response to a GO message). *<move>* is the content of the player's move.
2. **`QUIT`** - Informs the server that the client wants to quit the game. This message can only be sent as a response to a GO message. After receiving this message, the server should send back an END message.

After sending a GO message, the server awaits for a MOVE message and during this time it is blocked, i.e. the program doesn't continue before receiving an input. Hence, a player which doesn't response a GO message for more than 30 seconds will consider as they quit and be removed from the game (the server sends END message and stops listening).

If there is just one player left in the game, that player is declared the winner, and the game is over. Otherwise, the game continues without the players who have left or have been forced to leave (due to a timeout).

Note: The player (human user) should just type their move, for example, "4". The client program is then responsible for adding the string MOVE, as specified by the protocol, to the message sent to the server, resulting in the message "MOVE 4". If a player wants to quit the game, they should type **`quit`**, and the client program should send the server a QUIT message.

Protocol infringements:

You can't presume that messages are valid (i.e. following the protocol). If a client sends an invalid message, the server should disconnect the client by sending an END message. Similarly, if a client receives invalid message from the server, it should disconnect and terminate itself.

The rationale for this strict rule is that messages do not rely on user input, hence a client that does not follow the protocol is almost always a faulty client.

Some examples for an invalid messages:

MOV 3

HAVE abcd

Running the programs:

The *Game Server* Program:

You will need to implement a program called *game_server* which runs the server. The command to run the server should look as follows:

```
game_server <Port Number> <Game Type> <Game arguments>
```

For example:

```
game_server 4444 numbers 3
```

You may assume all parameters are valid.

If a problem prevents the game from continuing (no connection, server cannot use the port, etc.), the server should show an informative message beginning with the word ERROR, then close all resources, and terminate itself.

The *Game Client* Program:

You will need to implement a program called *game_client* which runs a client program that allows it to communicate with the server. Each player should run a *game_client* of its own. The command to run a client is:

```
game_client <Game Type> <Server Name> <Port Number>
```

For example:

```
game_client numbers mypc 4444
```

In the above example, the computer name on which the server is running is 'mypc'. You should replace the server name with the computer name on which you run your server.

If a problem prevents the game from continuing (no connection, server sends an invalid message, etc.), the client should close all its resources, and terminate itself.

If the player (human user) enters a **`quit`** command, the client should send a QUIT message to the server, and then close resources and terminate.

Handling errors:

In case the player enters wrong input (an input that doesn't follow the game rules), the server responds with an error message that starts with **`TEXT`** ERROR. The client should then print out the error message, i.e. everything after **`TEXT`**, including the word ERROR.

Note that this is not a protocol error, but a game error. As mentioned above, any protocol error follows by disconnecting from the game. After a game error the player has a chance to replay. However, to avoid an endless loop, a player (client) who performs 5 game errors in a row, is disconnected from the game.

The Game - Numbers

Numbers is a simple multi-players game (the following example shows the game with 2 players).

The rules:

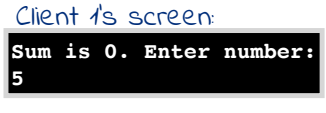
In their turn, each player chooses a number between 1 and 9.

The sum of all previously chosen numbers (of all the players together) is stored on the server, and is given to the player prior their turn.

The game is won by the first player to attain a total of 30 or more.

The following is an example of playing the game with 2 players, and including all messages sent over the connection:

Server to Client 1:	TEXT Sum is 0. Enter number:
Server to Client 1:	GO
Client 1 to Server:	MOVE 5
Server to Client 2:	TEXT Sum is 5. Enter number:
Server to Client 2:	GO



Client 2 to Server:	MOVE 15
Server to Client 2:	TEXT ERROR Bad input. Try again:
Server to Client 2:	GO
Client 2 to Server:	MOVE 9

Client 2's screen:

```
Sum is 5. Enter number:
15
ERROR Bad input. Try again:
9
```

NOTE: This is just an example. You may want to print out more details error messages, instructions, etc.

// after some turns	
Server to Client 1:	TEXT Sum is 21. Enter number:
Server to Client 1:	GO
Client 1 to Server:	MOVE 9
Server to Client 1:	TEXT You won!
Server to Client 1:	END
Server to Client 2:	TEXT You lost!
Server to Client 2:	END

Submission:

The following files should be uploaded to Learning@Griffith:

- **zip file** that includes your source code plus statically linked executable (include all makefiles, project files, project subdirectories, readme file; except the object files).
- **Software documentation** must be submitted according to the sample documentation that is available on learning@griffith. Please use the sample as a template and change the contents of each section to reflect your assignment.
- Software documentation must include compilation instructions (compilers used, dependencies, etc).
- readme file should include instructions on how to run the game (server and clients).