SE 3313 Updated Design Document
Battleship

Team members:

Stefan Kramreither (250747762)
Kirkland Landry (250730276)
Fabio Moná Torres (250737198)
Joseph Roussy (250739514)

Intro:

Battleship is a game between two players which provides both players with a 10x10 grid. In the first phase of the game, players set their ships on their side of the field. Each player has 5 ships: a patrol boat which occupies 2 squares, a submarine which occupies 3 squares, a destroyer which occupies 3 squares, a battleship which occupies 4 squares, and an aircraft carrier which occupies 5 squares. In the second phase of the game, players take turns firing shots on the other player’s grid. In the classic version of the game, the other player would verbally declare the result of the shot however in our implementation the software running at the client side will determine this information. A ship is considered “sunk” if the opponent fires shots on all the grid squares the ship occupies. The first player to sink all the opponent’s ships wins the game.

Client UI:

In the GUI for this battleship game the player will be able to join a game. Once a game is joined they will be able to place ships on a grid. This grid will only be able to place one of each type of ship by clicking the ship name and then clicking where they would like to place it on their grid. Any invalid positions will not be allowed and will be represented by red circles. The player will be able to rotate the ship by pressing the R key before placing. Players will not be able to place a ship that would intersect or be on top of another ship. Once both players have placed their ships and pressed a start button they will see an empty grid representing the opponents space. Players will take turns shooting at their opponent by clicking squares on the enemy grid, however a player can only make 1 shot per turn. Hits to the enemy will be represented on the enemy grid and hits to a player's ship will be represented on their own grid where they placed their ships. When a game is over both players will be notified of who the winner is. The client also has a text box which is updated along with the UI as events in each turn occur (such as a shot on a certain square or a ship being sunk). This acts as a log of all events in the game organized by round number.

Client Design:

We decided against the state design pattern because the states were not really interesting enough to justify the work. Instead we decided to simply count the hits on a ship by decrementing a counter.

As in the trademarked game, the aircraft carrier can take 5 hits before sinking, battleship 4, submarine and destroyer 3, and patrol boat 2.

Player shot locations are validated in the client. So, if a square has already been attacked, it cannot be attacked again. Once a valid square has been clicked, the log will show this attack and the coordinates will be sent to the server.

The player being attacked will get a message from the server telling it which square is being attacked. This attack will be displayed on the log regardless of hit or miss. If that square is occupied, the hit will be displayed on the attacked square and the ship that was hit must decrement its counter. How this is done is explained in the next paragraph. The client will update the local log ("hit!"), send an appropriate message to the server and update the ship's state. If the square is not occupied, the miss will be displayed on screen, the log will say "miss!", and a miss message will be sent to the server.

Each grid square has an associated ship code, set when ship is placed there. A hit results in a getShipCode() call which returns an int to be used in a switch statement. The appropriate case calls hitShipWithCode(*code*) which decrements the appropriate hit counter returns a bool. If bool is true, the ship has been sunk and the outgoing message is set to that ship name instead of ‘hit’. Otherwise, the basic ‘hit’ response is sent.

We simplified our message format. All messages contain a single word (e.g. 'hit', 'miss', 'win'), the name of a ship indicating that it has been sunk, or coordinates in the form 'x,y'. We used a function called logWrite() to translate client/server single word messages into a nice update for the player to read. This also required a few bool variables to track if it was your turn or not and also if you have the first turn in each round or not to update the round number in the log.

The attacking player's log will be updated and the square will change to show a hit or miss. Unless the game has ended, then the other player's turn will begin.

When the game ends for any reason, you must spawn a new game window from the first form opened. From the spawn window you may spawn multiple games and play against someone on the same computer if no one else on the network is waiting for a game. See Client-Server Interaction for more information on how people get matched up.

The turn log follows a format similar to the example below:

```
------------------------------------------------------------------------
        Round 1:
        Your move.
        You fired at 0,1 -- hit!
        Your opponent's move.
        Your opponent fired at 6,7 -- miss!
        ...
        Round x:
        Your move.
        You fired at 2,8 -- hit!
        Battleship was sunk!
        Your opponent's move.
        Your opponent fired at 3,1 -- hit!
        ...
        Round y:
        Your move.
        You fired at 6,9 -- miss!
        Your opponent's move.
        Your opponent fired at 4,3 -- hit!
        Patrol Boat was sunk!
        You Lose
------------------------------------------------------------------------
```

Client-Server Interaction:

It easiest to consider the design of the server in terms of the interaction between the clients and the server. The client-server interaction for our program will be designed around using sessions to manage the current game between two clients. The session will be stored on the server and will include two threads to hold the socket connection between the server and the two clients. The session will also include a binary semaphore for each client thread and two strings that will hold the current data from Socket.Write(). The interaction between the two clients and the server can be summarized in a number of parts:
Note: Client 1 = BLUE, Client 2 = RED.

Part 1: Create a session
BLUE starts up their client, opens a socket on the server and blocks on Socket.Read(). BLUE's server thread that contains the socket connection blocks using BLUE's SEMAPHORE. RED starts up their client, opens a socket on the server and blocks on a Socket.Read(). RED's server thread that contains the socket connection signals BLUE's SEMAPHORE and both threads perform Socket.Write("start") to signify the start of a game.

Part 2: BLUE makes a turn
RED performs Socket.Read() and RED's thread blocks on the server using RED's SEMAPHORE to wait for BLUE to make their turn. BLUE's thread on the server performs Socket.Read() and blocks which waits for BLUE to do its turn. BLUE performs its turn by performing, for example, Socket.Write("4,3") which contains the coordinates of the playing field it wants to attack and performs Socket.Read() which waits for RED's response to the attack. BLUE's server thread unblocks RED's SEMAPHORE and RED's thread performs Socket.Write("4,3") to RED's client and then performs Socket.Read() to RED's client to wait for RED's response on whether the attack was a hit or a miss. BLUE's thread blocks on BLUE's SEMAPHORE in the meantime. RED's client performs, for example, Socket.Write("hit") which contains the confirmation that an attack hit a ship, unblocks BLUE's SEMAPHORE, and BLUE's thread performs Socket.Write("hit") to notify BLUE's client the attack was a hit.

Part 3: Winning Condition for RED
When BLUE receives a hit so that all ships are sunk, it will perform Socket.Write("win") instead of Socket.Write("hit") to signify that RED has one the game. In this situation, the server will close the sockets of both players and delete their threads. BLUE will display a losing notification while RED will display a winning notification signifying the end of the game.

Part 4: Disconnects
If any of the clients disconnect from the server, the socket connection will close automatically and the actions described in Part 4 will occur.

Part 5: BLUE exits mid-game
If BLUE exists mid-game, the socket connection between BLUE and the server will close automatically. Therefore, the next time BLUE’s server thread tries to perform a Socket.Read(), it will return 0 and the server will send socket.Write(“close”) to RED’s client and close RED’s socket connection. RED’s client will then indicate that their opponent has left and that they won the game.

As such, in terms of connecting to a game, if there is no one currently waiting then a client will wait after searching for a game. If there is someone waiting and another client connects then they will be matched up and a session will begin. Note in this context, we could have multiple clients on the same machine since it is an instance of the game window that connects to the server and a single user can spawn many game windows. As such a split screen game can be played on the same machine or a user could play 2 games at once against two other users.

Reflections

The C++ server in our project deals with thread management, synchronization, and process termination by using sessions and semaphores. Since the game battleship only requires two clients to communicate with each other, the server uses sessions (analogous to a current game between two clients) that each contain two threads, two semaphores, and two strings. The session manages two threads which each contain a socket connection to a client and have access to each other's semaphores and strings. To synchronize access to the string objects, semaphores are used so that only one thread is running while writing to its string or reading from the other thread's string. On termination of the session, the session will signal both semaphores, close both sockets, and destroy both threads. The original design for Part 2 of this project included the session objects and therefore did not need to be modified beyond creating the threads, semaphores, and strings. Throughout creating the server, sessions, and threads we learned many things, especially how to use the socket class, blockable objects, and the thread class. Since there was only one thread class used for both threads in a session, it was interesting to write code as if the thread was communicating with itself but at a different section of the file.

The client side was written in C# so we had to learn the syntax for operating the sockets since we had only ever used sockets in Python. We made our client side in parts and then joined them so our first socket experience had a form with two textboxes and a button. When the form opened, the socket set up would run within a try/catch block. Clicking the button would convert the text from the first textbox into a byte array and ping Professor McIsaac's server. The response would be changed back to a string and displayed in the second textbox, and when the event of form closing fired, the socket would be closed. None of the socket work was very difficult though it was slightly different from our experience in Python. The libraries available in C# made the whole process pretty quick and painless. Most of the code from this mini project was added to the main project within functions called TCPWrite and TCPRead. We realized that some of our ideas in the initial design document were overkill and could be simplified so instead of a state design pattern we simply used bool variables and counters. Similarly, our messages were simplified in order to make the sending and receiving quicker.

Discussion

We found this project worthwhile since we were able to see the intricacies of designing our own TCP server that actually sent data between clients.

We learned how to use the C++ threads, semaphores, and sockets to design a server that was able to sort clients into 2 player sessions and complete a game of battleship. The knowledge of threads, semaphores, and sockets can be applied to running your own socket server with its own protocol for communicating with clients. The client server interaction and socket connections can also be applied to designing games. We learned how to use C# threads, delegates, and sockets to read from and write to a remote server. This choice to use Windows forms on the client side was based on our previous experience in C# and event-based programming.

We believe this project was of moderate difficulty with the information taught in class making the C++ server easy to implement. We found the layout of this lab perfectly fine and Part 1 of the lab actually helped us understand the C++ server code more and allowed us to finish Part 2 on the server side relatively quickly. We actually found the sending and receiving of the messages easier than implementing parts of the client side and making the log behave how we wanted it to based on the client-server messages.

We found this project very worthwhile. It was very interesting to create both the client and server side of an application. As with most other things, we learned much more through doing this lab then just having discussions in class. However the discussions in class did help us in the design of our game. In fact, the discussion about different approaches about spiting up the complexity of the application between the client and the server may have been more useful if it had been earlier in the class.

Our original idea would have been a real time game (we considered slime soccer and rock’em sock’em robots) however we changed to a turn based game based on the class discussion about how hard it would be to synchronize a real time game over the internet. This discussion almost certainly saved us from going down a path that was much more complicated than we first expected.

We also enjoyed the open ended nature of this assignment. Not only were we able to make any kind of game or chat application, but we could choose between the two. This allows us to exercise our creativity in ways that other assignments rarely do. We think this assignment should remain open ended in the future and not restrict groups to making only one kind of application.