Milestone 2 Report

Garrett Kalter

My code is a two-player (port) implementation of tic-tac-toe using sockets to communicate between the two players. Below are the descriptions of each function.

1. The **check_win** function checks if a player has won the game by looking at the current state of the Tic-Tac-Toe board.
2. The **send_board** function sends the current state of the board to a player's connection, replacing empty cells to make the board clear
3. The **play_game** function does all the gameplay, alternating between the two players. It prompts each player for their move, validates the input, updates the board, and checks for a win or tie condition. It then sends appropriate messages to the players about the game state.
4. The **handle_connection** function sends initial instructions to each player upon connection.
5. In the **main** function:
   - A socket is created and bound to a specific host and port (in the 5000s).
   - The server listens for incoming connections from players.
   - Once two players have connected, the game begins.
   - The game loop runs until a win or tie condition is met, and then the connections are closed, and the server socket is closed.
6. In the future, this will either be hosted on a server or a library of games.

Cade Smith

My Code is a two player server based implementation of the game hangman that uses sockets and threads to allow two players to play against each other. Below is a brief overview of my code:

1. The main() function controls the server. The server has a capacity of two connections, and maintains a dictionary called game_state and global variable called current_turn
   a. The current turn variable is initially set at 1, meaning that the player that connects first will go first.
2. If a player connects, the handle_client function is called with that socket at the player's assigned number (1 or 2 depending on when the player connected)
3. The handle client function then runs a check to see if it is the players turn
   a. If it is not, the server sends a message and ignores input from the player
   b. If it is, the server performs another check

        i. If a hangman word has not been provided, the server asks for a word
        ii. If a word has been provided, the current guessing template is displayed and the player is allowed to guess a letter
            1. If a letter has already been guessed, then the server prompts for another word.
            2. If a letter has not been guessed, then the message for either a correct or incorrect guess is displayed. In either case the "hangman" and "guessed letters" are also listed
        iii. The current turn changes to the other player
4. Game plays until either a word is guessed or one of the players hangmen is "hung"

Goals for final milestone:

- Add cases to make sure the provided word is valid?
    - Could use NLTK library
- Find more elegant way of handling turn state
- Find better implementation for game state

Lucca Figlioli - Battleship

My code is a 2 player implementation of the classic Battleship board game using Sockets in Python. The game requires a server to be running, and two separate users (either on different machines or terminals) to connect to this server in order for the game to begin.

Server:

1. Server main(): The main method in the server is used to create a new instance of BattleshipServer with a hard coded host address, port number, a socket, and an array for the players in the game. It then calls the start method on that server.
2. BattleshipServer.start(): The start method within the BattleshipServer class first binds the socket to the host and port number, then listens for new connections. The server then waits for new connections and adds the client sockets to the "Player" array until the size of the array is 2. The start method then calls the setup_game() method.
3. BattleshipServer.setup_game(): This method first creates a new BattleshipGame with a hardcoded board size, player boards as 2D arrays, player ships as arrays of tuples, and sets of tuples for each player that represent their correct guesses. This method then calls setup_board() on each of the player sockets.
4. BattleshipServer.setup_board(): This method sends a message to the client passed as a parameter and accepts ship placements from that client in the form (X,Y). The

BattleshipGame.place_ship() method is called to check if the coordinates are valid. This process is repeated until 5 valid ships are placed by the client.

5. BattleshipGame.place_ship(): This method is used to determine whether or not the client has submitted usable coordinates to the server for their ships. This includes checking for repeat ship placement and attempting to place a ship out of bounds. If these checks pass, the ship's coordinates are added to the appropriate array of tuples.
6. BattleshipServer.setup_game() CONT: After each client has placed their ships, each client is given alternating turns (starting with player 1) to guess the locations of the other player's ships. If a client hits a ship, they are given the message "Hit!" and an X is placed at that location on their board. If they miss, they are given the message "Miss!" and a 0 is placed at that location on their board. This is repeated until one of the players has correctly guessed the locations of all five of the other player's ships.
7. BattleshipServer.format_board(): This method is simply to convert the 2D array that represents the game boards to a string that can be sent to the clients.

Client:

1. Client main(): The client main method simply creates a new BattleshipClient object with its own host, port, and socket. The socket is then connected to the server socket using the BattleshipClient.connect() method. The main method then calls BattleshipClient.setup()
2. BattleshipClient.setup(): This method is used in conjunction with the BattleshipServer.setup_board() method to prompt the client for where they want to place their ships. These coordinates are then sent over the sockets to the server where they are either verified or rejected. Once 5 ships have been verified, this method calls the BattleshipClient.play() method to move onto the next stage of the game.
3. BattleshipClient.play(): This method operates similarly to the setup() method, but is repeated until there is a winner. The method constantly checks for the winning message and whether or not it is that client's turn to determine how many messages to accept from the server and when to send coordinates back as that client's move. This method is not written very elegantly, but it required extensive debugging to get the timing and ordering right for the client/server communication.

Future Work:

I hope to improve the error handling and detection of this game for the final milestone. As of right now, if the client does not enter coordinates correctly, the server will not handle the errors and simply shutdown. This should be an easy fix. I also hope to implement this game on an actual server that contains the rest of the games created by our group members. This will allow players to select games from an online library and will allow players to compete on separate devices instead of just on different terminal windows.

James Burke - Connect4

I began working on my code by creating a functional copy of my existing C code from my 2D Grid Game project from last semester in 208. I was able to utilize similar logic for checking win conditions, switching players, and initializing the board. After I developed the basic game, I began working on a version that incorporates sockets so that the game can be played remotely across two terminals. What followed was a process of trial and error that resulted in me changing my implementation multiple times, ending up with me rewriting much of the code that I had developed as a framework in the first place.

Methods:

1. send_board(self, conn): This function sends an updated version of the game board across a socket, row by row. I initially used a function called print_board that returned a string version of the game board across a socket, but this ended up causing errors when running the game.
2. check_win(self): This function calls upon much of my original logic for checking a winner from my original implementation of Connect4. This function is called every time a player makes a move and checks for vertical, horizontal, or diagonal adjacency through indexing the neighbors of a given position on a 2D grid.
3. make_move(self, row, col): This method attempts to place the current player's token in the specified column. It begins at the bottom of the column and works its way up to find the first available spot. If successful, it updates the board and returns True. If the column is full or the index is out of bounds, it returns False. This method ensures that only valid moves are made during gameplay.
4. switch_player(self): A straightforward method that toggles the current player between 'X' and 'O'. This is called after each valid move to alternate turns between the two players.
5. play_game(conn1, conn2): Orchestrates the game flow using two socket connections, one for each player. The method initiates the game, continuously sends the game board to both players, and requests and validates their moves. Upon receiving a valid move, it checks for a win condition or a tie. If there's a winner, it notifies both players of their status and ends the game. If all positions are filled without a winner, it declares a tie and ends the game.

6. main(): Sets up the server socket, binds it to an IP and port, and listens for connections from two players. Once both players are connected, it calls play_game to start the session. After the game concludes, it closes all connections and shuts down the server.

Future Work:

I plan on doing some more comprehensive gameplay testing to check for bugs. Additionally, we are going to compile all of the games to be played together within a unified library hosted on a dedicated server. We may even consider adopting a basic API if this does not prove to be too cumbersome.