Pong Project

Brooke McWilliams, brmc230@uky.edu
Morgan Miller, mdmi240@uky.edu

Background:

The client-server Pong game is designed to allow two players to play Pong over a network. The primary goal is to establish communication between the client and server to relay and receive information about the game state. The server manages the game state (This includes paddle locations, ball position, and the current score) and communicates with two clients simultaneously. The clients handle user input, update their local game state, and send updates to the server. The process involved dealing with new concepts like threading, synchronization, and socket programming.

Design:

Client Design:

The client script utilizes the Pygame library to create a Pong game interface. It communicates with the server, sending paddle location and receiving information about the opponent's paddle, ball, and score. The initial client code, presented in **pongClientV1.py,** served as a starting point but was later deemed redundant and evolved through subsequent versions. The final client implementation **pongClientV3.py** effectively communicates with the server, sending paddle locations, and receiving updates on the opponent's paddle, ball, and score.

Server Design:

The server code, located in **pongServer.py**, handles communication with two clients simultaneously using sockets and threads. It sends initial game specifications to each client and creates a separate thread for handling updates from each client. It relays information about the opponent's paddle, ball, and score to the clients. Two semaphores are used to ensure exclusive access to the game state and avoid data corruption. The server code underwent multiple versions for optimal functionality. The initial version **pongServerV1.py** included redundant server-side logic that was later refined. The subsequent version **pongServerV2.py** represented a partially complete iteration of both server and client code. The final server implementation **pongServerV3.py** emerged as a fully functional solution, managing simultaneous communication with two clients using threading and ensuring synchronization.

Implementation:

Client Implementation:

The client initializes a socket connection to the server, communicates its paddle location, and receives updates on the opponent's paddle, ball, and score. The **playGame** function manages the game loop, handling player actions, ball movement, and updates are sent to the server. Threading is utilized to ensure a responsive interface.

Server Implementation:

The server initiates socket connections, listens for incoming client connections, and handles communication with two clients simultaneously, sending initial game specifications to each. Threading is utilized to manage concurrent communication. Semaphores ensure thread safety when updating the game state. The server sends and receives updates to and from the clients to keep the game synchronized.

Challenges:

- Synchronization: Coordinating game state updates between the server and clients using threading and semaphores created challenges. The server handles synchronization by comparing the sync counters of clients and updating the lagging client's game state. Ensuring that updates from both clients are processed accurately required was something that required a lot of attention for us.

- Networking Challenges: Handling network communication posed challenges, especially in synchronizing game states between clients and the server. Making sure everyone sees the game in the same way was tough. One issue was figuring out the right server host and IP. It took a lot of trial and error to find an appropriate server host and IP to ensure smooth connectivity between clients and the server. Getting the server host and IP right was extremely important for the game to run smoothly.

Lessons Learned:

- Threading and Semaphores: Understanding and implementing thread-safe mechanisms, such as semaphores, was essential when dealing with shared data among multiple threads. It allowed for synchronization and increased the ability in managing simultaneous tasks.

- Networking Communication: Handling network communication introduced complexities, especially in synchronizing game states between clients and the server. This helped us learn more about how to handle these network challenges, which enhanced our ability to troubleshoot and advance this crucial aspect of the client-server architecture.

- Socket Programming: We learned how to set up communication between the server and clients. The pongServer.py file demonstrates our use of the socket module to create a server socket, associate it to a specific host and port, and handle multiple clients concurrently through threading. On the client side, socket programming allowed us to establish a smooth connection with the server, exchanging game-related information.

Conclusions:

This project provided experience creating a multiplayer Pong game with a client-server architecture. Overall, synchronization challenges and understanding how to set up communication between the server and clients served as our biggest problems. The resulting implementation (V3) combats those challenges and successfully facilitates gameplay between two clients.