CS371
Tag Howard
Juliann Hyatt
John Michael Stacy
November 17, 2023

Pong Project

Background:

The project aimed to create a Pong game using Python with additional features such as network functionality, account registration, spectation, replayability, and an HTML scoreboard. The design included all the extra credit options, and the team went through approximately 8 different iterations with an emergency fallback Pong game design.

The client plays a crucial role in facilitating communication with the server in the context of the Pong game. It is tasked with relaying and receiving vital information about the current state of the game. Primarily, the client is responsible for sending data pertaining to the location of the user's Pong paddle to the server. Conversely, it receives essential updates from the server, including details such as the location of the opponent's paddle, the current position of the ball, and the ongoing game score.

The server manages simultaneous communication with two clients using sockets and threads. It relays critical game information, such as paddle positions, ball location, and score, ensuring seamless interaction. Acting as the central hub, the server synchronizes clients with the dynamic game state, crucial for handling the interactive nature of the Pong game effectively.

Design:

Before implementing the code, we designed our Pong Server to enable multiple games to run at the same time while allowing different clients to connect to our system. In essence, we planned to have multiple server sockets binding at different ports that allowed encryption. The goal was for clients to join a Lobby server and from there login or create an account. Once validated and logged in, the client can then join a game using a code or create a game.

After a client creates the game (or joins), the client disconnects from the Lobby server and a Game Process is initiated to create a new server. The client then joins that new Game Server tied to a special game code. From there, the Game Server handles the entire game from connection requests to replaying and managing wins in our database. Once a game is complete, the clients have the option to replay by pressing “r” on the keyboard. We thought of error handling and planned to create a class to control the socket connection for every user of the system.

Implementation:

1. Connection Process:

Clients initiated connections by sending login requests to the lobby server. Unverified clients are promptly denied access, while verified clients may opt to create or join a game with a unique code. Joining a game involves the transmission of a join request with the game code to the server.

2. Game Setup:

Creating a game involves the server generating a game object (via multiprocessing) and a unique code. Clients then leave the Lobby Server and join the Game Server using a unique port generated during Game Setup. Once the Game Server is created and running it accepts incoming

clients and Handles the following Request: Start, Grab, and Update. These request from clients inform the Game Server when to update its game information, when to request the Start of the game, and Grab the current game model.

3. Game Execution:

During the game loop, after each iteration, both clients sent their sync numbers and game states to the server. In the event of a discrepancy, the server transmitted the game state with the highest sync number to both clients, effectively synchronizing the game state. At the conclusion of the game, clients signaled the end by sending a stopped game request to the server. Should both players decide to replay, a new game request was sent to the server, initiating the game from scratch. In the event that a player leaves, an error message will pop up and once you click done you will return to your Tkinter Main Menu in the main Lobby window to join another game or start another game. Once a game completes, users can restart without leaving the Game Server.

4. Leaderboard:

On port 80, we have a leaderboard that you can access via a browser (leaderboard.html) to highlight the top players. On port 8500 we serve an API that pulls the leaderboard data from SQLite and is called by JavaScript on the leaderboard.

Challenges:

Race conditions posed a significant obstacle, demanding careful management during the game state update. Additionally, the team encountered issues stemming from multiple revisions, particularly concerning game names. Since the project spanned over two months, sometimes pieces of code were not touched for weeks at a time, and variable names got switched around. Some team members navigated a learning curve, grappling with Python and pygame for the first time. Efficient time coordination for collaborative work and the temptation to invest substantial

time in bonus features, even when the base code was yet to function optimally, added further complexity to the project. The tiered bonus point structure proved to be another pain point, as we had the most trouble with the play again feature after already implementing security and scoreboards.

High ping (across a VPN for example) remains an issue for us as the round trip (RTT) over a poor connection can be upwards of 100ms. This means that when one machine is close to the server (less than 25ms RTT) and the other has a high ping, each game loop can take many times longer on the far machine. This is because we synchronously send and receive a message from the server on every tick of the client. We could work around this by shifting more game/physics logic to the server or by making our calls asynchronous, but either option would have significantly increased the complexity of our game loop.

Known Issues:

While the main process of the game is bug-free (to our knowledge), there are some edge cases and odd behavior that could cause some issues. Most of these revolve around what happens when the server shuts down or ends a game (because one of the players left). Because of the way the synchronous connections work and the fact that we did not implement a robust shutdown message the client usually just times out or the socket fails. This usually leads to a popup and a graceful shutdown, but in some cases (like if the client was in the middle of an operation) the game will freeze up for a moment before actually closing properly. While the client does gracefully handle not being able to reach the server on startup, there is no retry implemented. In fact if the client ever fails to read from the server, or if it times out, it assumes the world has ended and the connection is unrecoverable.

Lessons Learned:

The project significantly enhanced the team's programming skills in Python, pygame, and Tkinter. Collaborative development, facilitated by extensive use of GitHub, became a cornerstone of our approach, as it allowed us to work separately but still keep up-to-date with everyone else, and to efficiently get patches into all versions of code.

Conclusion:

The project provided valuable insights into Python, pygame, Tkinter, and collaborative development. The challenges faced, particularly the race conditions and multiple revisions, underscored the complexities of networked game development. The extensive use of GitHub allowed us to get more experience with the program and to add to our own repositories. Special acknowledgment is extended to John for his unwavering dedication and engineering prowess demonstrated throughout the project.