CS371 Fall 2025 - Project Report

Pong Multiplayer Game - Client/Server Architecture

Authors

Jayadeep Kothapalli – jsko232@uky.edu

Harshini Ponnam – hpo245@uky.edu

Rudwika Manne – rma425@uky.edu

Background

The goal of this project was to build a complete real-time multiplayer Pong game using a client-server architecture. The problem we needed to solve was how to keep multiple remote players synchronized in a fast, interactive game while maintaining fairness and preventing desynchronization. Unlike the traditional single machine Pong game, where we only use one PC, we aim to create a networked version that must handle latency, inconsistent client performance, packet delays, and the possibility of unreliable or slow connections. To solve these challenges, the game must centralize all decision-making, physics, scoring, and winning conditions on a single authoritative server.

In addition to real-time synchronization, the project required secure user authentication, encrypted communication, and persistent storage of user data and win history. This introduced a second problem: how to design a lightweight, secure authentication system that does not slow down gameplay. We addressed this by implementing password hashing with PBKDF2-HMAC and symmetric encryption with Fernet.

Finally, the project needed to support spectator clients, rematch coordination (both players must agree to play again), and an online leaderboard accessible through a simple HTTP server. Each of these features introduces further technical challenges in concurrency, shared state management, and structured communication protocols.

Overall, the main problem we solved was designing a reliable, secure, real-time game system where multiple players and spectators interact smoothly over the network, while ensuring consistent gameplay and persistent tracking of user progress.

Design

Before writing any code, we outlined the system as a set of coordinated components that communicate through a well-defined protocol. Our design centered around one key principle: the server is authoritative, and all clients act only as input devices and display units. This ensured fairness, prevented cheating, and avoided desynchronization between the players.

System Architecture

We began by identifying the roles of each part of the project:

Server:

- Maintains the authoritative game state
- Updates ball physics, paddle movement, and score of the players
- Coordinates rematches (requires both players to press R key on the keyboard)
- Handles user registration and login
- Encrypts or decrypts gameplay messages

- Tracks wins and serves a persistent leaderboard over HTTP
- Allows any number of spectators

Client:

- Displays the game using Pygame
- Sends player movement and “ready” signals to the server
- Receives game state continuously in a background thread
- Provides a Tkinter-based Join UI and a CLI fallback
- Encrypts outgoing gameplay messages

This separation defined clear responsibilities and reduced complexity during development.

Communication Protocol

We designed a simple, readable, line-based protocol:

Server -> Client

For each frame, the server sends either plaintext or encrypted:

<leftY> <rightY> <ballX> <ballY> <lScore> <rScore>\n

- Encrypted for players
- Plaintext for spectators

Client -> Server: "up", "down", "", "ready"

Each message is encrypted using Fernet for left and right players.

This strict protocol made debugging easier and ensured that both sides always agreed on the state format.

Concurrency Model

The server uses three threads:

- Thread 1: Main game loop
- Thread 2: Handle left player’s encrypted input
- Thread 3: Handle right player’s encrypted input

Plus, an additional thread for:

- Accepting spectator clients
- Running the HTTP leaderboard server

On the client side:

- Main thread: Game loop and rendering

- Background thread: Receiving state from server

This design prevents blocking and keeps gameplay consistently at 60 FPS.

Rematch System

We designed a synchronized rematch mechanism:

- After a win, the server enters “win state”
- Both players must send "ready"
- Only then does the server reset scores and ball/paddles

This ensures fairness and prevents one client from forcing an immediate restart.

Authentication and Encryption Design

Security was built as a separate module (security.py):

- Passwords stored as salted PBKDF2-HMAC hashes
- User data stored in JSON for simplicity
- All gameplay messages encrypted with Fernet symmetric encryption
- Registration and login done over plaintext commands before encryption begins

We designed the auth first so we could integrate it cleanly before movement or game logic.

Leaderboard Design

The leaderboard uses:

- A JSON file (leaderboard.json) for persistence
- A simple HTTP server on port 80 that
    - Generates an HTML table
    - Displays player initials and total wins

We designed this to run in a daemon thread, so it does not interfere with gameplay.

Implementation

Our implementation followed the design principles established earlier: the server is fully authoritative over the game world, while clients act only as input devices and render engines. We divided the system into three coordinated modules: the server, the client, and the security layer. Each module was implemented with concurrency and real-time communication in mind.

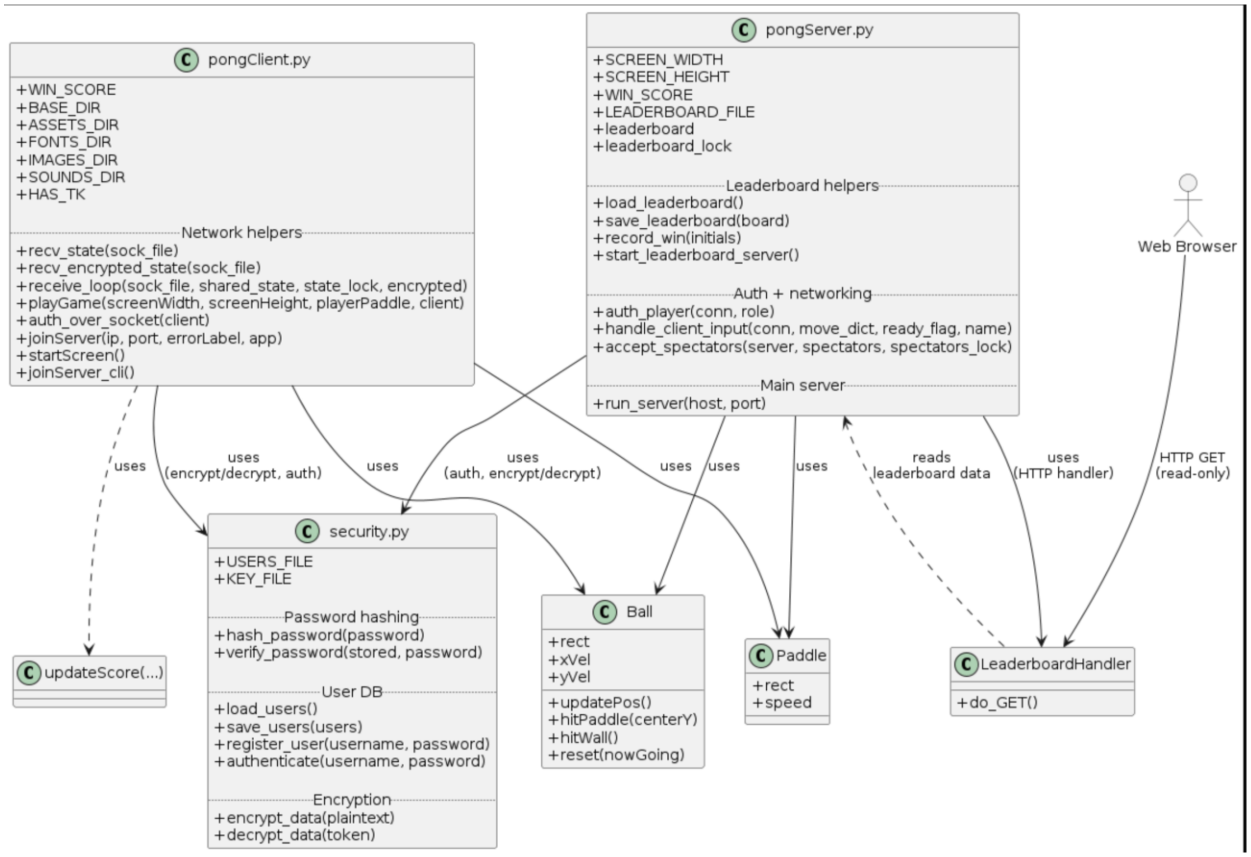

Figure 1 – High-level UML design of our Pong system. The pongServer.py module acts as the authoritative game server and uses Paddle and Ball for game logic, security.py for authentication and Fernet encryption, and LeaderboardHandler to serve the persistent leaderboard over HTTP. The pongClient.py module uses the same game objects for rendering, calls security.py for encrypted gameplay, and uses updateScore(...) from the helper code to draw scores. A web browser interacts only with LeaderboardHandler via HTTP GET requests to view the leaderboard.

Server Implementation

The server (pongServer.py) maintains complete, authoritative control of the Pong game. It handles authentication, per-frame game logic, encrypted message processing, spectator support, and persistent leaderboard storage.

a. Authentication Phase

When a new client connects, the server enters a message-based authentication handshake. Communication is simple, human-readable, and easy to debug.

Pseudocode for Authentication Loop

```
function auth_player(socket):
    send("AUTH <role>")
    repeat:
        line = receive_line(socket)
        (cmd, user, pass) = parse(line)


        if cmd == "register":
            if register_user(user, pass):
                send ("OK registered")
                return user
            else:
                send ("ERR username exists")


        else if cmd == "login":
            if authenticate (user, pass):
                send ("OK logged-in")
                return user
            else:
                send ("ERR invalid credentials")
```

This process continues until valid registration or login occurs.

b. Input Handling Threads

To avoid blocking the main game loop, we created separate threads for each player's encrypted inputs. Each thread receives Fernet tokens, decrypts them, and updates shared dictionaries representing the latest paddle movement and rematch readiness.

Pseudocode for Client Input Thread

```
thread handle_client_input(socket, shared_move, shared_ready):
    buffer = ""
    loop:
        buffer += socket.recv()
        while newline in buffer:
            token = extract_line(buffer)
            message = decrypt(token)
            if message in {"up", "down", ""}:
                shared_move.value = message
            else if message == "ready":
                shared_ready.value = True
```

This keeps the input path responsive even if a client experiences slight delays.

c. Game Loop (60 FPS, Authoritative)

The main game loop advances at 60 FPS, performing all physics, score updates, win detection, and rematch coordination. It is the heart of the system.

Pseudocode for Core Game Loop

```
loop every 1/60 second:
    update_paddle_positions(left_move, right_move)

    if someone reached WIN_SCORE:
        if both_ready: reset_game()
    else:
        update_ball_position()
        detect_collisions()
        detect_scoring()

    state = format_state(leftY, rightY, ballX, ballY, lScore, rScore)
    send_encrypted(state, left_player)
    send_encrypted(state, right_player)
    send_plain(state, spectators)
```

This loop ensures consistency across all clients, no matter their hardware or network conditions.

d. Spectators & Leaderboard

A dedicated thread accepts unlimited spectators. They receive only plaintext state updates and do not affect the game.
Meanwhile, a lightweight HTTP server publishes a live leaderboard stored in leaderboard.json.

Pseudocode to Record Win

```
function record_win(initials):
    lock(leaderboard)
    leaderboard[initials] += 1
    write_to_file(leaderboard)
    unlock(leaderboard)
```

Client Implementation

The client (pongClient.py) acts as the rendering interface and input controller. It runs both local and network logic simultaneously.

a. Connection & Role Assignment

The Tkinter UI (or CLI fallback) establishes a TCP connection, reads the server's initial configuration, and learns the role: left player, right player, or spectator.

b. Authentication Flow (Player-Only)

Players perform an interactive text-based login/registration over the socket.
Spectators skip authentication entirely.

Pseudocode for Client Authentication

```
read_intro()
repeat:
    ask user: register or login?
    send("<cmd> <username> <password>")
    response = read ()
until response starts with "OK"
```

c. Encrypted Client Movement

Clients encrypt movement commands using the shared Fernet key. Spectators send only empty lines since they do not control a paddle.

Pseudocode for Send Movement

```
loop:
    if key_down(UP):     move = "up"
    else if key_down(DOWN): move = "down"
    else: move = ""

    if spectator:
        send(move)
    else:
        send(encrypt(move))
```

d. Background Receiver Thread

To avoid blocking the Pygame loop, a background thread listens for incoming state lines and updates a shared dictionary.

Pseudocode for Receive State Thread

```
thread receive_loop(socket, shared_state, encrypted):
    loop:
        if encrypted:
            token = read_line()
            plaintext = decrypt(token)
        else:
            plaintext = read_line()

        state = parse(plaintext)
        lock(shared_state)
        shared_state = state
        unlock(shared_state)
```

This keeps rendering smooth and latency low.

e. Rendering Loop

The main Pygame loop reads the shared state and draws the entire game at 60 FPS:

- paddles
- ball
- scores
- win messages
- logo

The client never computes physics; it simply displays what the server instructs.

Security Module Implementation

The security.py module provides password hashing, encrypted communications, and key management.

a. PBKDF2-HMAC Password Storage

Passwords are salted with 16 random bytes and hashed with PBKDF2-HMAC using SHA-256 and 200,000 iterations.

Pseudocode for Hashing

```
function hash_password(password):
    salt = random_bytes(16)
    hash = PBKDF2(password, salt, 200000)
    return salt || hash
```

Pseudocode for Verification

```
function verify (stored, password):
    salt = stored [0:16]
    stored_hash = stored [16:]
    check_hash = PBKDF2(password, salt, 200000)
    return stored_hash == check_hash
```

b. Fernet Encryption

The system uses a single shared Fernet key for gameplay encryption. The key is stored in fernet.key and loaded by security.py, so the server and all clients use the same symmetric key to encrypt and decrypt movement and state messages. This guarantees that different devices can communicate securely without having to exchange keys at runtime.

Pseudocode - Encryption

```
function encrypt(text):
    return Fernet(key). encrypt(text)
```

c. Persistent JSON Files

Two persistent JSON files are maintained:

- users.json - usernames + hashed passwords
- leaderboard.json - initials + win counts

This provides simple, cross-platform data durability.

Overall, our implementation followed the original design closely and resulted in a stable, fully functional multiplayer Pong system. The server controls all game logic, authentication, encryption, and leaderboard functionality, while clients focus on input and rendering using a background network thread to maintain 60 FPS gameplay. By separating concerns across modules and using threaded communication with careful state management, we were able to build a smooth, secure, real-time game that supports players, spectators, rematches, and persistent user data.

Challenges

Throughout the development of our multiplayer Pong system, several parts of the project did not go as planned. Many of these challenges came from the fact that real-time networked games behave very differently from simple local programs. Each issue required us to rethink our design or adjust our implementation.

Handling Real-Time Networking Without Lag or Freezes

Our first major challenge was managing real-time communication over TCP. Initially, the client attempted to read state updates directly in the main Pygame loop. This caused intermittent freezing whenever the server delayed sending data even for a few milliseconds.

How we adapted:

We introduced a dedicated background receiver thread to handle all incoming state messages asynchronously. This allowed the game loop to run smoothly at 60 FPS regardless of network latency.

Synchronizing Rematches Without Desynchronization

Implementing the “both players must press R to rematch” feature was more complex than expected. At first, only one client’s ready signal was being detected properly, causing one player to restart while the other stayed in game-over mode.

How we adapted:

We added shared dictionaries (left_ready, right_ready) protected by simple flags and made the server check:

```
if left_ready and right_ready:
    reset scores and paddles
```

This guaranteed that the game resets only when both sides explicitly agree.

Encryption Broke Communication Temporarily

When we first encrypted messages using Fernet, we accidentally encrypted all outgoing messages including initial config lines that the client needed to parse as plaintext. This caused the client to fail immediately with decoding errors.

How we adapted:

We restructured communication into two phases:

1. Pre-authentication: plaintext only
2. Post-authentication gameplay: encrypted for players, plaintext for spectators

Clear separation between these phases fixed the inconsistency.

Dealing With Multi-Threading Race Conditions

During testing, paddle movement and score updates sometimes appeared to “jump.” This was caused by threads modifying shared state (for example left_move["value"]) while the game loop was reading it.

How we adapted:

We simplified the shared variables to minimal dict structures and ensured reads/writes were atomic enough for Python’s GIL. This eliminated visual stutter without requiring heavy locks.

Persistent Leaderboard File Corruption

While building the leaderboard, the JSON file occasionally became corrupted if the server was closed abruptly in the middle of writing.

How we adapted:

We added a dedicated function save_leaderboard() and protected all writes using a threading lock, ensuring only one write operation occurs at a time.

Managing File Permissions for Fernet Key and Leaderboard Server

Running the HTTP leaderboard server on port 80 caused permission issues on Windows because port 80 requires administrative privileges. Additionally, the Fernet key file could not be created due to write permissions in some directories.

How we adapted:

We shifted to running the server from elevated permission terminals and placed the .key file in the project directory where we knew we had written access. This stabilized key creation and HTTP server startup.

Testing Multiplayer Locally Was Surprisingly Hard

Running two clients on one machine led to several issues:

- Shared Pygame audio clashing
- Windows security system prompts blocking sockets
- Port conflicts between multiple server restarts

How we adapted:

We adjusted our workflow:

- Always start the server first
- Launch two separate terminal windows for client instances
- Disable audio on one instance if needed
- Explicitly close servers between runs

This allowed consistent testing.

Summary

Most challenges came from real-time networking, concurrency, and encryption order-of-operations. Each issue required redesigning small portions of the system or restructuring threads. The final implementation is significantly more stable because of these adaptations, and these challenges helped us better understand how real multiplayer systems work.

Lessons Learned

This project taught us several concepts that we did not fully understand before starting. While Pong seems simple, building a real-time, multiplayer, encrypted, client-server version requires us to learn practical skills that go far beyond typical classroom exercises.

Real-Time Networking Is More Complicated Than Basic Socket Examples

Before this project, we had only used sockets for simple request-response programs. We learned that real-time games cannot block, and that network delays can break gameplay. This taught us:

- Why non-blocking design is essential
- Why game networking always uses background receiver threads
- How to maintain shared game state safely between threads

This was our first experience building a system that needs to run at 60 FPS while receiving data continuously.

The Importance of Having an Authoritative Server

We knew the theory behind client-server architecture, but we never had to enforce it.
We learned that:

- Letting clients compute physics leads to cheating and desynchronization
- A single trusted server eliminates most consistency problems
- Clients should be pure “dumb terminals” that only send input and render visuals

This helped us understand why real games (Fortnite, Rocket League, etc.) work the way they do.

Secure Password Storage Is More Than Just Hashing

Before this project, we did not realize how insecure plain hashing was.
We learned about:

- PBKDF2-HMAC for slow, salted hashing
- Why random salts prevent rainbow-table attacks
- Ensuring never to store user passwords in plaintext

We now understand the basics of safe authentication design.

Encryption in Networked Games Requires Strict Ordering

We assumed encrypting messages would be simple.
We learned the hard way that:

- Some messages must remain plaintext (config, registration)
- Encryption must start after login
- Both sides must agree exactly when encryption begins

Any mismatch leads to unreadable data and broken connections.
This taught us practical protocol design discipline.

Proper Separation of Concerns Makes Large Projects Maintainable

We experienced first-hand why dividing a system into modules matters:

- security.py for hashing/encryption
- pongServer.py for game logic + networking
- pongClient.py for UI, input, and rendering

Without this structure, debugging would have taken far longer.

Concurrency Is Not Just a Theory

We had to work with:

- Server threads for left/right players
- A background thread for accepting spectators
- A background thread on the client for receiving states
- A thread for the HTTP leaderboard server

We learned how threads can conflict, how shared state can break when not handled carefully, and how to design simple, safe data structures for real-time code.

Persistence and File Handling Require Care

Before this project, writing JSON files felt trivial.
We learned:

- File writes can corrupt if they overlap
- Leaderboards require consistent read/write cycles
- Authentication and history files must not be overwritten accidentally

This gave us practical experience while maintaining persistent data safely.

Testing Multiplayer Systems Requires a Different Workflow

We discovered that:

- Two clients on one PC behave differently than two clients on two machines
- Security system settings, ports, and threads all complicate local testing
- Multiplayer debugging requires patience and structured test cases

This was our first experience testing a distributed system.

Lesson we learned overall:

We now understand that even a simple game becomes a complex engineering problem when real-time networking, concurrency, encryption, and persistent data are involved. The project taught us how to think like systems designers rather than just programmers, and it showed us how small design decisions can affect the entire architecture.

Known Bugs

At this time, we do not know of any reproducible bugs in the core gameplay, authentication, encryption, or client-server communication when running in the expected environment (Python 3.x with Pygame, Tkinter, and cryptography installed).

The main limitations are environment-related:

- The HTTP leaderboard may require admin/root privileges on some systems to bind to port 80.

Conclusions

This project allowed us to build a complete and reliable real-time multiplayer Pong game using a fully authoritative client–server architecture. By combining networking, concurrency, encryption, authentication, and persistent storage, we created a system that behaves like a simplified version of real-world online games. The server maintains full control of the game state; players exchange encrypted inputs, spectators can join at any time, and wins are recorded permanently and displayed through an HTTP leaderboard.

Throughout development, we applied systems-level thinking: separating concerns across modules, designing a clear communication protocol, and using thread-based concurrency to keep networking and rendering smooth at 60 FPS. Implementing secure password storage with PBKDF2-HMAC and encrypted gameplay messages with Fernet strengthened the reliability and safety of the system. Our rematch coordination and spectator support added additional layers of synchronization and robustness.

The final system meets all project requirements and demonstrates how even a simple game becomes a complex engineering problem when real-time networking and security are involved. This project gave us practical experience in scalable architecture design, protocol development, and multi-threaded programming skills that extend far beyond the scope of Pong and are fundamental for building modern distributed applications.

The final system was tested with two physical devices plus an additional spectator client, confirming that synchronization, authentication, and rematch logic worked reliably.