CS371 Networked Pong Project Report

Ghaleb Abualsoud

Novemeber 2025

1 Background

The purpose of this project was to create a real time, two-player networked Pong game. The core problem addressed was how to synchronize two remote players so that both experience identical game behavior. Because Pong requires fast updates and consistent physics, the system needed a design that prevented desynchronization, supported continuous communication, and maintained smooth gameplay. The project also aimed to apply fundamental networking concepts such as client server communication, message serialization, and socket programming.

2 Design

Before implementing the code, the design began with the decision to use a centralized server that would maintain the authoritative game state. All physics updates including ball movement, paddle collisions, and scoring were planned to run exclusively on the server to avoid discrepancies between clients. The clients would perform only local rendering and input handling, making them lightweight and easier to manage.

TCP was chosen for communication because of its reliability and simplicity. Messages between the server and clients were planned to be serialized using Python's pickle module, allowing flexible data structures to be exchanged with minimal overhead.

A basic game loop was designed as follows: each client would send its paddle position to the server, and the server would respond with the updated game state for rendering. The user interface would run in Pygame, while networking would operate concurrently. The system also included a simple replay mechanism: once a player reached the winning score, both clients would be prompted to decide whether to play again.

3 Implementation

The design was implemented in three primary Python modules: the server script, the client script, and a helper file containing game object classes. The server accepts exactly two clients, assigns them to the left and right sides, initializes the game objects, and enters a synchronized game loop.

Below is pseudocode that reflects the implemented server structure:

```
# Server setup and matchmaking
server_socket = socket.socket()
server_socket.bind((HOST, PORT))
server_socket.listen(2)

left_client, _  = server_socket.accept()
right_client, _ = server_socket.accept()

# Initialize game objects
ball = Ball()
left_paddle  = Paddle()
right_paddle = Paddle()
left_score  = 0
right_score = 0

game_running = True

while game_running:

    # Receive paddle movement from clients
    left_paddle_y  = recv(left_client)
    right_paddle_y = recv(right_client)

    # Update ball motion
    ball.update()

    # Wall collision
    if ball.hits_vertical_wall():
        ball.reverse_y()

    # Paddle collisions
    if ball.hits_left_paddle(left_paddle_y):
        ball.reverse_x()
    if ball.hits_right_paddle(right_paddle_y):
        ball.reverse_x()

    # Scoring logic
    if ball.out_left():
        right_score += 1
        ball.reset()
    elif ball.out_right():
        left_score += 1
        ball.reset()

    # Package and send game state
    state = {
        "ball_x": ball.x,
        "ball_y": ball.y,
        "left_paddle_y": left_paddle_y,
        "right_paddle_y": right_paddle_y,
        "left_score": left_score,
        "right_score": right_score,
        "game_over": (left_score == WIN or right_score == WIN)
    }

    send(left_client, state)
    send(right_client, state)

    if state["game_over"]:
        game_running = False
```

The following pseudocode shows the implementation of the clients:

```
# Connect to the server
client_socket = socket.socket()
client_socket.connect((HOST, PORT))

pygame.init()
window = pygame.display.set_mode((WIDTH, HEIGHT))

running = True

while running:

    # Player input
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    if player_side == "left":
        if key_pressed(W):
            paddle_y -= SPEED
        if key_pressed(S):
            paddle_y += SPEED
    else:
        if key_pressed(UP):
            paddle_y -= SPEED
        if key_pressed(DOWN):
            paddle_y += SPEED
    # Send paddle update
    send(client_socket, {"paddle_y": paddle_y})

    # Receive updated state
    state = recv(client_socket)

    # Draw frame
    draw_ball(state["ball_x"], state["ball_y"])
    draw_left_paddle(state["left_paddle_y"])
    draw_right_paddle(state["right_paddle_y"])
    draw_score(state["left_score"], state["right_score"])

    pygame.display.update()

    if state["game_over"]:
        running = False
```

The replay phase was implemented as follows:

Server Replay Logic

```
send(left_client,  {"prompt": "play_again"})
send(right_client, {"prompt": "play_again"})

left_ans  = recv(left_client)["answer"]
right_ans = recv(right_client)["answer"]

if left_ans == "yes" and right_ans == "yes":
    restart_game()
else:
    close_sockets_and_exit()
```

Client Replay Logic

```
msg = recv(client_socket)

if msg["prompt"] == "play_again":
    choice = get_user_input("Play again? (y/n): ")

    if choice.lower() == "y":
        send(client_socket, {"answer": "yes"})
    else:
        send(client_socket, {"answer": "no"})
        exit_game()
```

4 Challenges

One challenge was ensuring synchronization between clients. Since both players rely entirely on the server's updates, any delay or missed message could cause visual stuttering or temporary freezes. Another challenge was separating networking logic from rendering logic, as Pygame requires precise event handling while networking requires constant I/O operations. Balancing both without blocking the game loop required careful structuring.

A further challenge involved managing disconnections or unexpected input. While the program handles normal gameplay smoothly, detecting abnormal termination and shutting down gracefully required additional logic.

5 Lessons Learned

This project clarified the importance of centralizing state in multiplayer games. Previously, it might have seemed viable for each client to compute its own physics, but this project demonstrated how quickly desynchronization can occur without a single authoritative source. Additionally, working with TCP sockets reinforced the need to manage blocking I/O carefully and to serialize data effectively.

Another key insight was recognizing how game loops must be designed to interleave rendering, input handling, and networking consistently. This project also provided practical experience with concurrency, serialization, and debugging distributed systems.

6 Known Bugs

Occasionally, if a client disconnects abruptly, the server may freeze or wait indefinitely for input. Given more time, this could be resolved by adding timeout checks or non-blocking socket reads. Another minor issue is that paddle movement may appear slightly jittery under network delay; smoothing techniques or interpolation could alleviate this.

7 Conclusions

This project successfully implemented a fully functional two-player networked Pong game. The server maintains the authoritative physics and score logic, while the clients handle user interaction and rendering. The design proved effective for maintaining synchronization, and the implementation provided valuable experience with network programming and real time game loops. While there is room for improvement, particularly in error handling and visual smoothing, the core system performs reliably and demonstrates the essential principles of client server game architecture.