Implementing Multiplayer in Tic Tac Toe

https://github.com/kh1z3r/multiplayer-tictactoe

Group UG13-4007:

Khizer Butt, Fareed Fareed-Uddin, Pedro Mantese Masegosa, Juan Martinez, Kevin Rapkin

03/28/2025

CNT4007 Communication Networks

Semester Spring 2025

Dr. Mohammad Ilyas
Florida Atlantic University

IMPLEMENTING MULTIPLAYER IN TIC TAC TOE

Written by Fareed-Uddin, Khizer Butt, Kevin Rapkin, Pedro Mantese, Juan Maritnez

---

Abstract:

This paper presents a modern implementation of networked multiplayer Tic Tac Toe using Python and Pygame, featuring both single-game and best-of-three match modes. The system employs a client-server architecture over TCP/IP networks using port 5555, enabling real-time remote play with responsive user interaction. Our implementation includes a comprehensive game state management system that handles player turns, move validation, and score tracking across multiple rounds. The client application provides an intuitive interface with a cream-colored theme, animated elements, and multi-modal feedback through both visual and audio cues. Technical contributions include a modular board system for game state representation, efficient network communication through JSON serialization, and robust error handling for network disconnections. The architectural design clearly separates client and server responsibilities: the server manages game logic, player connections, and match state, while the client handles rendering, user input, and real-time chat functionality. Quality-of-life features include dynamic waiting screens, turn indicators, win tallies for best-of-three matches, and immediate visual feedback for player actions. The implementation demonstrates the successful integration of networking principles with game development techniques, providing a foundation for future extensions such as additional game modes and enhanced social features.

1. Introduction

This project implements a networked multiplayer Tic Tac Toe game that demonstrates fundamental concepts in network programming and real-time game development. The primary objectives include creating a responsive multiplayer experience, implementing robust game state management, and providing social features like in-game chat.

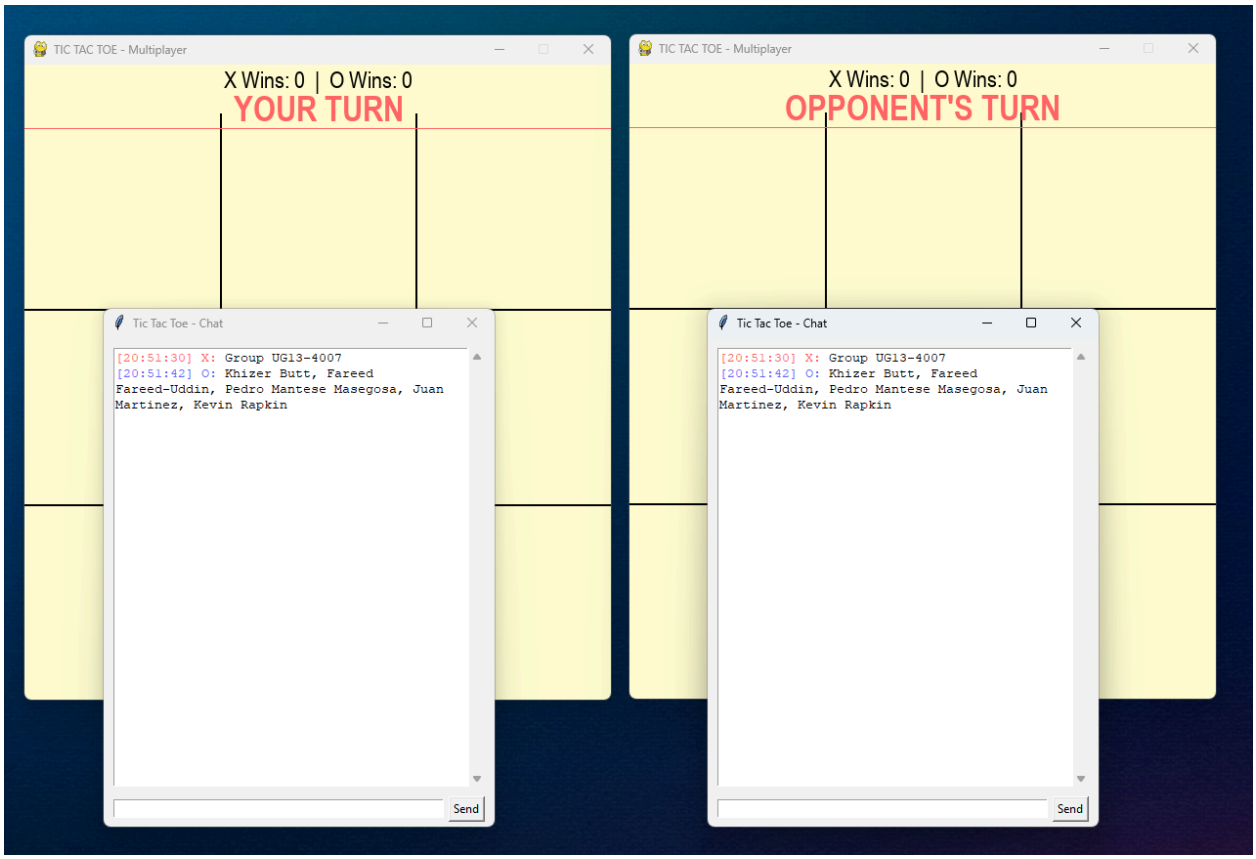
The game showcases several key networking concepts including TCP/IP communication, client-server architecture, and real-time data synchronization. The implementation uses socket programming to establish reliable connections between players, with the server acting as a central coordinator for game state and player interactions.

```python
class TicTacToeServer:
    def __init__(self, host='0.0.0.0', port=5555):
        self.server = socket.socket(socket.AF_INET, SOCK_STREAM)
        self.server.bind((host, port))
        self.server.listen(2)
        self.clients = []
        self.current_player = "X"
        self.game_mode = "single_game"  # default game mode
```

server.py*

Tic Tac Toe serves as an ideal platform for demonstrating these concepts due to its simple rules but complex implementation requirements. The game extends beyond basic gameplay to include features such as:

- Best-of-three match system
- Real-time chat functionality
- Dynamic turn indicators
- Audio-visual feedback
- Score tracking



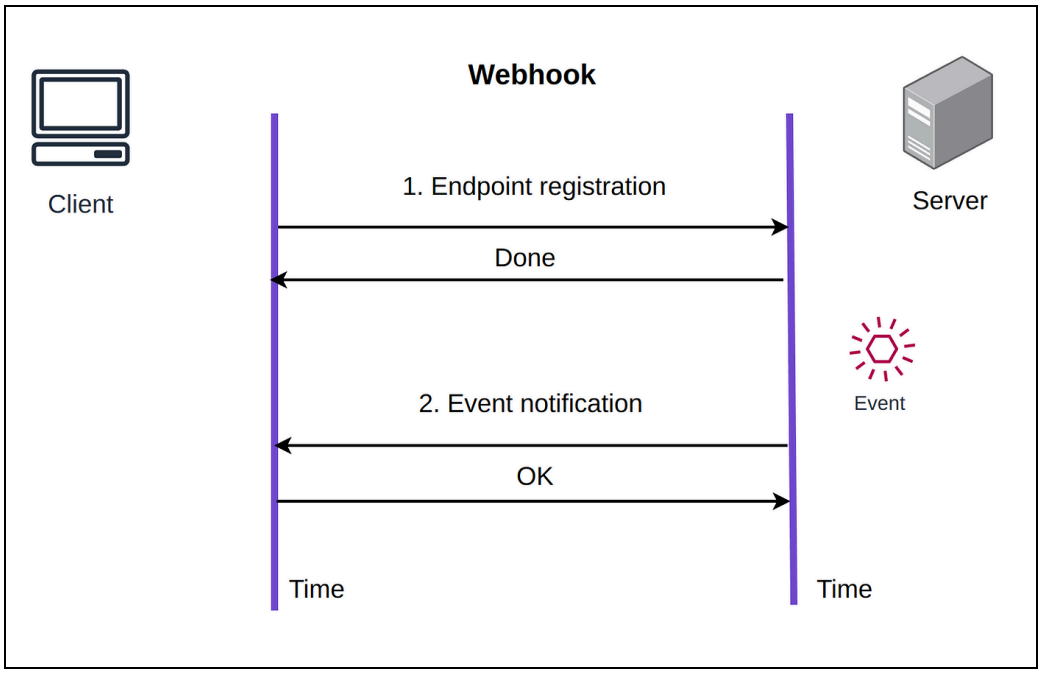
2. System Architecture

Client-Server Model

The system employs a traditional client-server architecture where the server manages game logic and state while clients handle user interaction and display. This separation ensures consistent gameplay across different network conditions and prevents cheating by validating all moves server-side.

```
class NetworkGame:
    def __init__(self):
        self.surface = pygame.display.set_mode(WINDOW_SIZE)
        self.board = Board()
        self.player_symbol = None
        self.current_player = "X"
        self.client = None
        self.chat_window = None
```

→ client.py



Client-Server Communication Flow

Component Interaction

The system consists of three main components:

1. Server: Handles game logic, player connections, and state management
2. Client: Manages rendering, input processing, and local state
3. Board: Represents game state and validates moves

```
class Board:
    def __init__(self):
        self.grid_matrix = [["" for x in range(3)] for y in range(3)]
        self.switch_user = False
        self.gameover = False
```

Technologies and Libraries

The implementation utilizes:

- Python 3.6+: Core programming language
- Pygame: Graphics and input handling
- Socket: Network communication

- JSON: Data serialization
- Threading: Concurrent operations
- Tkinter: Chat window interface

Design Decisions and Trade-offs

Key architectural decisions include:

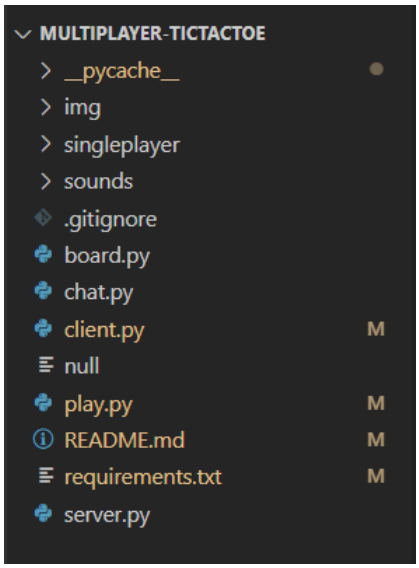
1. Using TCP over UDP for reliable game state transmission
2. Implementing a separate thread for chat functionality to prevent gameplay interruption
3. Centralizing game logic on the server to ensure consistency
4. Using JSON for data serialization to maintain compatibility and readability

```python
class TkChatWindow:
    def __init__(self, send_message_callback):
        self.thread = threading.Thread(target=self.create_window)
        self.thread.daemon = True
        self.thread.start()
```

Repository Structure

The project maintains a clean separation of concerns:

multiplayer-tictactoe/
client.py # Client implementation
server.py # Server implementation
board.py # Game board logic
chat.py # Chat system
play.py # Game launcher
requirements.txt # Dependencies
img/ # Game assets



The architecture prioritizes modularity and extensibility, allowing for future additions such as:

- Player profiles and authentication
- Match history tracking
- Enhanced social features
- Additional game modes

This design has proven effective in creating a responsive and engaging multiplayer experience while maintaining code maintainability and system reliability

3. Network Infrastructure

The network infrastructure of our Tic Tac Toe implementation utilizes TCP/IP sockets for reliable communication between clients and the server. This section details the key components of our networking implementation.

Socket Implementation

The server initializes a TCP socket that listens for incoming connections on port 5555:

```python
class TicTacToeServer:
    def __init__(self, host='0.0.0.0', port=5555):
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.bind((host, port))
        self.server.listen(2)  # Maximum 2 players
        self.clients = []
        print(f"Server started on {host}:{port}")
        print(f"Your IP address is: {socket.gethostbyname(socket.gethostname()
```

→ server.py

The client establishes a connection to the server using similar socket programming

```python
def connect_to_server(self, host):
    try:
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client.connect((host, PORT))
        self.connected = True
        thread = threading.Thread(target=self.receive_messages)
        thread.daemon = True
        thread.start()
        return True
    except Exception as e:
        print(f"Connection error: {e}")
        return False
```

→ client.py

Protocol Design

The game implements a JSON-based protocol for all network communications. This ensures readable, extensible message passing between clients and server. Key message types include:

1. Game State Updates

```
self.broadcast({
    "type": "update_board",
    "board": self.board,
    "position": [x, y],
    "player": player
})
```

→ server.py

2. Chat Messages:

```
self.broadcast({
    "type": "chat_message",
    "player": player,
    "text": data["text"]
})
```

→ server.py

3. Game Control Messages

```
client.send(json.dumps({
    "type": "game_over",
    "winner": player
}).encode())
```

→ server.py

Connection Management

The server implements sophisticated connection management through several key mechanisms:

1. Client Tracking

```
def start(self):
    while True:
        client, address = self.server.accept()
        if len(self.clients) < 2:
            print(f"Connection from {address}")
            self.clients.append(client)
            self.addresses.append(address)
            thread = threading.Thread(target=self.handle_client,
                                      args=(client, address))
            thread.daemon = True
            thread.start()
```

→ server.py

2. Concurrent Client Handling:

Each client connection is managed in a separate thread to ensure responsive gameplay and prevent blocking operations.

4. Error Handling and Recovery

The implementation includes robust error handling mechanisms

1. Connection Loss Detection:

```
def remove_client(self, client):
    if client in self.clients:
        index = self.clients.index(client)
        self.clients.remove(client)
        self.addresses.pop(index)
        client.close()
        print(f"Client disconnected. {len(self.clients)} clients remaining."
```

2. Game State Recovery

```
if len(self.clients) < 2:
    if not (self.game_mode == "best_of_3" and self.game_over):
        self.board = [['' for _ in range(3)] for _ in range(3)]
        self.current_player = "X"
        self.game_over = False
```

3. Message Transmission Error Handling

```
def send_move(self, position):
    if self.connected:
        try:
            self.client.send(json.dumps({
                "type": "move",
                "position": position
            }).encode())
        except:
            self.connected = False
```

The network infrastructure is designed to handle common failure scenarios:

- Server unavailability
- Client disconnections
- Network latency
- Message corruption

- Port availability issues

This robust implementation ensures a smooth multiplayer experience while maintaining game state consistency across all connected clients. The system's error handling capabilities allow for graceful degradation in case of network issues and provide clear feedback to users when problems occur.

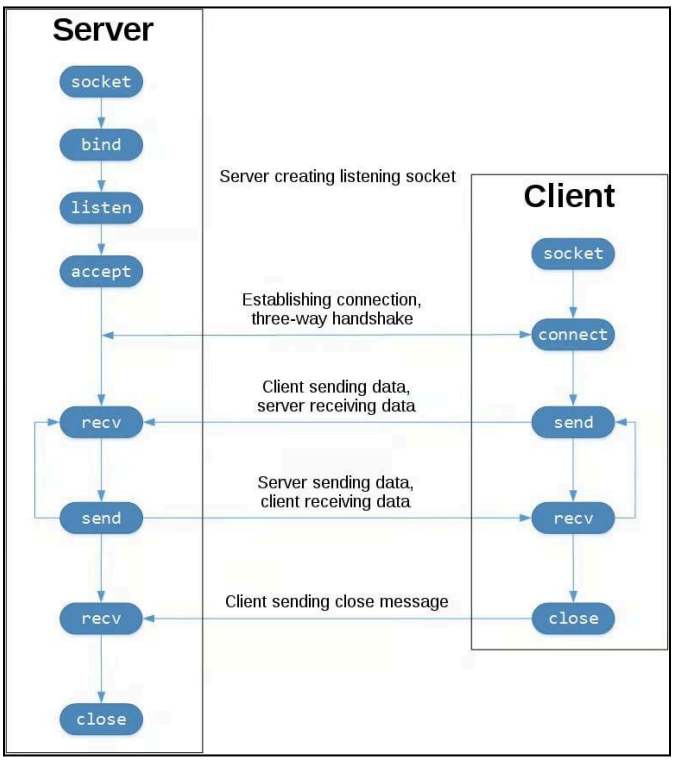
5. Project Results

The implementation successfully created a networked multiplayer Tic Tac Toe game with integrated chat functionality. The project demonstrates practical applications of network programming concepts using Python's socket library and concurrent programming through threading.

Network Implementation

The core networking functionality utilizes Python's socket library for TCP/IP communication

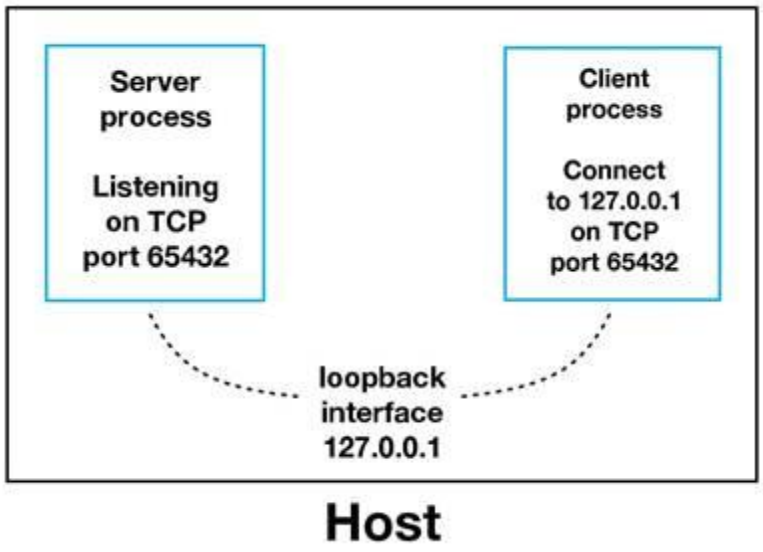
```
class TicTacToeServer:
    def __init__(self, host='0.0.0.0', port=5555):
        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.bind((host, port))
        self.server.listen(2)  # Maximum 2 players
        self.clients = []
        print(f"Server started on {host}:{port}")
        print(f"Your IP address is: {socket.gethostbyname(socket.gethostname()
```



Socket connection flow between client and server]

The server implementation supports both local and remote

gameplay through configurable IP addressing:



Loopback Interface Diagram - Which Shows local connection setup

User Interface and Game Modes

The game features a clean, quick terminal based intuitive interface with two game modes:

```
        TIC TAC TOE LAUNCHER


What would you like to do?
1. Start Server & Play (Host a game)
What would you like to do?
1. Start Server & Play (Host a game)
1. Start Server & Play (Host a game)
2. Join Game (Connect to a host)
3. Exit

Enter your choice (1-3): 1

Starting server...
Server started on 0.0.0.0:5555
Your IP address is: 192.168.5.96

Select game mode:
1. Single Game
2. Best of Three
Enter choice (1-2): 2

Starting game in best_of_3 mode...
pygame 2.6.1 (SDL 2.28.4, Python 3.13.1)
Hello from the pygame community. https://www.pygame.org/contribute.html
Connection from ('127.0.0.1', 7876)
Game mode set to: best_of_3
```

Game state updates are handled through JSON-encoded messages:

```
def send_move(self, x, y):
    """Send move to server"""
    if self.connected and not self.game_over and self.current_player == self.player_symbol:
        try:
            self.client.send(json.dumps({
                "type": "move",
                "position": [x, y]
            }).encode())
        except:
            self.connected = False
```

Concurrent Processing

To prevent UI freezing during network operations, the implementation uses threading:

```
def start(self):
    while True:
        client, address = self.server.accept()
        if len(self.clients) < 2:
            print(f"Connection from {address}")
            thread = threading.Thread(target=self.handle_client, args=(client, address))
            thread.daemon = True
            thread.start()
```

UNIT TESTING

Unit Testing was an important process during development of this game in order to verify the functionality of various components such as the X and O icons, sound effects, user interface, game logic, and network communications. Integration Testing was used to make sure that the separate systems like Singleplayer, Multiplayer, Best of 3, and Chat were able to be fully integrated into a seamless single experience. Going forward, Load Testing could be used to see how well the online components perform under stress and User Acceptance Testing could gather feedback on how satisfactory the end-user finds the final product.

6. Data Communication and Threading

Local vs Remote Connections

The server implementation supports both local and remote gameplay through configurable IP addressing. By default, the server binds to:

```
def __init__(self, host='0.0.0.0', port=5555):
    self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.server.bind((host, port))
```

For local gameplay, clients can connect using the loopback address (127.0.0.1), while remote players can connect using the host's public IP address. This flexibility is reflected in the client's connection interface:

```
def connect_to_server(self, host):
    try:
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client.connect((host, PORT))
```

Data Transmission

The game implements JSON-based message passing for all game actions. For example, when a player makes a move:

```
def send_move(self, x, y):
    """Send move to server"""
    if self.connected and not self.game_over and self.current_player == self.player_symbol:
        try:
            self.client.send(json.dumps({
                "type": "move",
                "position": [x, y]
            }).encode())
```

Threading Implementation

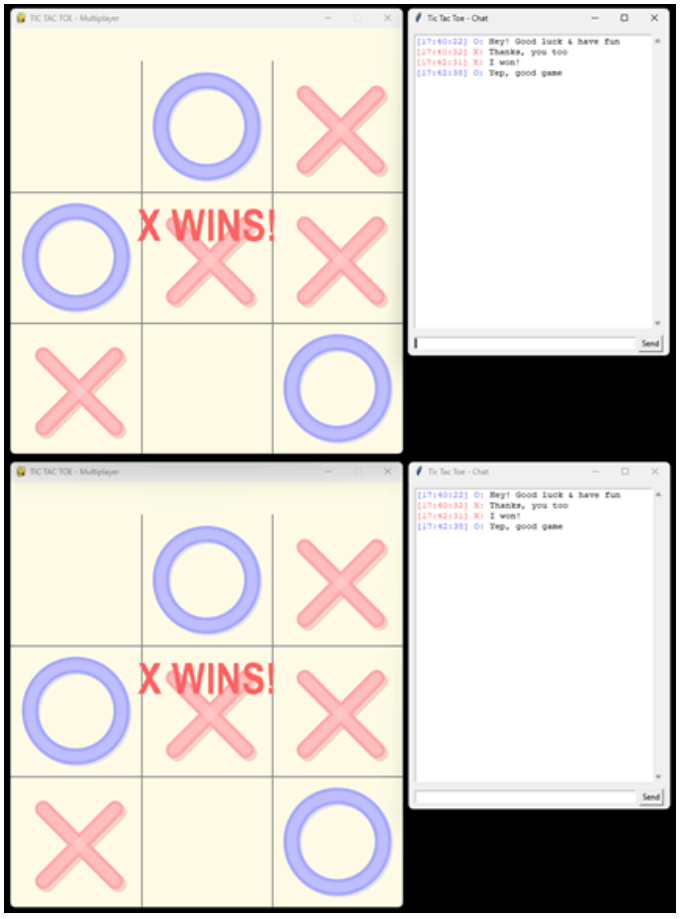
To prevent UI freezing during network operations, the game uses multiple threads:

1. Server Connection Thread:

```
def start(self):
    while True:
        client, address = self.server.accept()
        if len(self.clients) < 2:
            thread = threading.Thread(target=self.handle_client,
                                      args=(client, address))
            thread.daemon = True
            thread.start()
```

2. Chat Window Thread

```
class TkChatWindow:
    def __init__(self, send_message_callback):
        self.thread = threading.Thread(target=self.create_window,
                                       args=(send_message_callback,))
        self.thread.daemon = True
        self.thread.start()
```



The threading implementation is particularly crucial for the chat system, as it ensures:

- Continuous message processing without blocking the game
- Responsive UI for both game and chat windows
- Real-time updates for game state and chat messages

The screenshot demonstrates the successful integration of these components, showing:

- A completed game state (X wins)
- Active chat window with message history
- Synchronized game state between players
- Color-coded chat messages for different players

This implementation allows for seamless gameplay while maintaining active communication channels between players, demonstrating effective use of threading for handling multiple concurrent operations.

6. Messaging system

When it comes to chat functionality specifically, the chat system is implemented through a dedicated TkChatWindow class (in chat.py) that runs in a separate thread from the main game. This separation is crucial as it allows players to chat without interrupting the game flow, similar to how you might have a phone conversation while playing a board game in person.

The chat window is implemented using Tkinter, Python's standard GUI library, and operates asynchronously through a producer-consumer pattern. When a player sends a message, the following sequence occurs:

1. The player types a message and hits enter or clicks send in the chat window

```
def send_message(self, callback):
    """Send a message from the input field"""
    text = self.input_text.get().strip()
    if text:
        callback(text)
        self.input_text.delete(0, tk.END)
```

2. The client formats and sends a JSON-formatted message to the server:

```
def send_chat_message(self, text):
    """Send chat message to server"""
    if self.connected and text:
        try:
            self.client.send(json.dumps({
                "type": "chat_message",
                "text": text
            }).encode())
        except Exception as e:
            print(f"Error sending chat message: {e}")
            self.connected = False
```

3. The server processes and broadcasts the message to all clients:

```
def broadcast(self, message):
    """Send message to all connected clients"""
    for client in self.clients:
        try:
            client.send(json.dumps(message).encode())
        except:
            self.remove_client(client)
```

One particularly elegant aspect of the chat system is its message queuing mechanism. Instead of trying to update the GUI directly (which could cause thread safety issues), messages are queued and processed periodically. This is implemented in the TkChatWindow class through the check_queue method, which runs every 100 milliseconds:

```
def check_queue(self):
    if self.queue:
        self.chat_display.config(state='normal')
        for msg in self.queue:
            timestamp = msg['timestamp']
            player = msg['player']
            text = msg['text']

            # Set message color based on player
            if player == "X":
                tag = "player_x"
                color = "#ff6464"  # Red for X
            elif player == "O":
                tag = "player_o"
                color = "#6464ff"  # Blue for O
            else:
                tag = "system"
                color = "#646464"  # Gray for system
```

The chat system includes visual differentiation between players, using color coding (red for Player X and blue for Player O) to make it easy to follow the conversation. This is achieved through Tkinter tags, which are applied to each message based on the sender's identity.

To prevent message corruption during transmission, the client implements a sophisticated buffer system that handles partial JSON messages:

```
def receive_messages(self):
    """Handle messages from server"""
    buffer = ""
    while self.connected:
        try:
            chunk = self.client.recv(1024).decode()
            if not chunk:
                break

            buffer += chunk

            # Process all complete messages in the buffer
            while buffer:
                try:
                    json_end = buffer.find("}")
                    if json_end == -1:
                        break
                    json_end += 1
                    message = buffer[:json_end]
                    buffer = buffer[json_end:]  # Remove processed message
```

This buffering system ensures that messages are properly reconstructed even if they arrive in fragments, which is crucial for maintaining chat coherence in networks with varying latency or packet sizes.

7. Game State Synchronization

State Representation

The game state is maintained through a centralized server model using a matrix-based board representation:

```
class TicTacToeServer:
    def __init__(self, host='0.0.0.0', port=5555):
        self.board = [['' for _ in range(3)] for _ in range(3)]
        self.current_player = "X"
        self.game_mode = "single_game"
        self.player1_wins = 0
        self.player2_wins = 0
        self.player_symbols = {}  # Maps clients to their symbols
```

The game uses a centralized server model where the server acts as the single source of truth for all game states. This architecture ensures consistency across clients by maintaining:

- A 3x3 matrix representing the game board
- Current player turn tracking
- Game mode state (single game or best-of-three)
- Player win counts and round numbers
- Player symbol assignments

This centralized approach prevents state divergence between clients and simplifies conflict resolution, as all game-changing decisions are validated and broadcast by the server.

Update Mechanisms

The server implements a broadcast mechanism to ensure all clients receive state updates:

```
def broadcast(self, message):
    """Send message to all connected clients"""
    for client in self.clients:
        try:
            client.send(json.dumps(message).encode())
        except:
            self.remove_client(client)
```

Game moves trigger state updates through JSON-encoded messages:

```
self.broadcast({
    "type": "update_board",
    "board": self.board,
    "position": [x, y],
    "player": player
})
```

The server employs a robust broadcast system that ensures reliable state synchronization across all connected clients. When any game state changes occur:

1. The server validates the change request
2. Updates its internal state
3. Broadcasts the change to all connected clients
4. Handles any failed transmissions by removing disconnected clients

This mechanism is particularly important for maintaining game integrity during critical moments like:

- Move placements
- Turn transitions
- Win/draw declarations
- Score updates in best-of-three mode

Latency Handling

The implementation uses several strategies to handle network latency:

1. Asynchronous Message Processing:

```
def connect_to_server(self, host):
    try:
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client.connect((host, PORT))
        self.connected = True
        # Start thread to handle server messages
        thread = threading.Thread(target=self.receive_messages)
        thread.daemon = True
        thread.start()
```

2. State Validation:

```
def is_valid_move(self, x, y):
    """Check if move is valid"""
    return 0 <= x < 3 and 0 <= y < 3 and self.board[y][x] == ''
```

The implementation includes several sophisticated strategies to handle network latency and ensure smooth gameplay:

1. **Asynchronous Message Processing:**

- Client-side message handling runs in a separate thread
- Prevents UI freezing during network operations

- Enables responsive gameplay even with network delays

2. **State Validation:**

- All moves are validated both client-side and server-side
- Prevents illegal moves from desyncing the game state
- Handles edge cases like duplicate moves or out-of-turn attempts

3. Connection Management:

- Robust handling of client disconnections
- Automatic game state preservation
- Graceful recovery mechanisms for reconnection

Consistency Maintenance

The server maintains game consistency through several mechanisms:

1. Move Validation

```
if self.current_player == player:
    x, y = data["position"]
    if self.is_valid_move(x, y):
        self.board[y][x] = player
        # Broadcast update
```

2. Turn Management:

```
def remove_client(self, client):
    if len(self.clients) < 2:
        if not (self.game_mode == "best_of_3" and self.game_over):
            self.board = [['' for _ in range(3)] for _ in range(3)]
            self.current_player = "X"
```

The server employs multiple layers of consistency checks to maintain game integrity:

1. Move Validation:

- Ensures moves are only accepted from the current player
- Validates move coordinates against board boundaries
- Prevents overwriting of existing moves

- Maintains turn order integrity

2. Turn Management:

- Strict enforcement of alternating turns
- Clear indication of current player
- Automatic turn transitions after valid moves
- Handling of interrupted turns due to disconnections

7.5 Turn-based Synchronization Challenges

The implementation addresses several synchronization challenges:

1. Player Turn Enforcement:

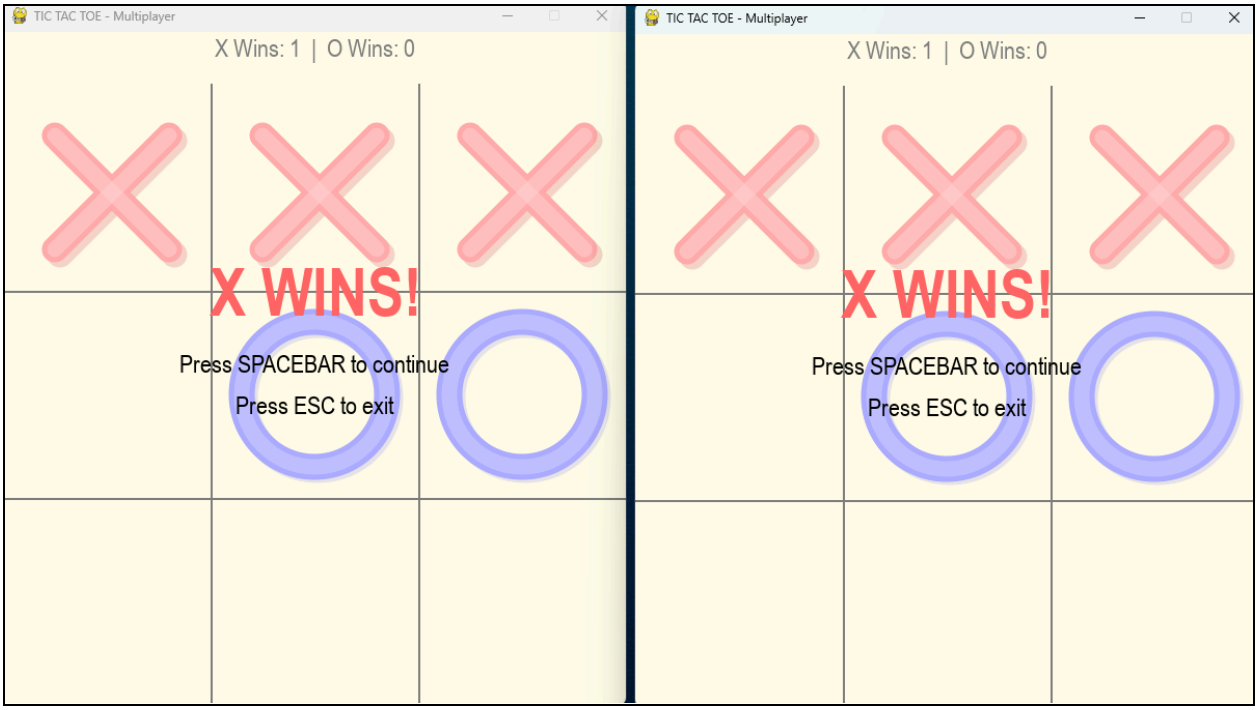
```
def send_move(self, x, y):
    if self.connected and not self.game_over and self.current_player == self.player_symbol:
        try:
            self.client.send(json.dumps({
                "type": "move",
                "position": [x, y]
            }).encode())
```

2. Game Mode Synchronization:

```
elif data["type"] == "game_mode":
    if self.player_symbols[client] == "X":  # Only host can change mode
        self.game_mode = data["game_mode"]
        self.broadcast({
            "type": "update_game_mode",
            "game_mode": self.game_mode
        })
```

3. Score Tracking in Best-of-Three Mode:

```
if self.game_mode == "best_of_3":
    if player == "X":
        self.player1_wins += 1
    elif player == "O":
        self.player2_wins += 1
    self.broadcast({
        "type": "game_of_3_over",
        "winner": player,
        "round_num": self.round_num,
        "player1_wins": self.player1_wins,
        "player2_wins": self.player2_wins,
    })
```



The implementation addresses several complex synchronization challenges:

Player Turn Enforcement:

The game implements a robust turn management system that maintains the integrity of the gameplay experience. When a player attempts to make a move, the system first validates their identity against the current turn state. This validation occurs both on the client and server side, creating a dual-layer protection against out-of-sequence moves. For example, if Player O attempts to move when it's Player X's turn, the move is immediately rejected without affecting the game state. The system also provides clear visual feedback to players about whose turn it is, reducing confusion and preventing accidental out-of-turn attempts. During network delays, the turn system maintains its state, preventing any potential race conditions where both players might act simultaneously. This is particularly important in situations where network latency might cause move information to arrive slightly delayed.

Game Mode Synchronization:

The game's mode synchronization system centers around a host-controlled configuration that ensures all clients maintain consistent game rules throughout

the session. When a host selects between single game or best-of-three modes, this choice propagates to all connected clients through a dedicated synchronization message. This ensures that all players operate under the same ruleset and scoring system. The system is particularly sophisticated in best-of-three mode, where it must maintain state across multiple rounds while ensuring that game rules, win conditions, and scoring remain consistent. When transitioning between rounds, the system coordinates the reset of the game board, turn order, and score display without requiring players to reconnect or restart the application.

Score Tracking in Best-of-Three Mode:

The score tracking system in best-of-three mode implements a comprehensive state management approach. It maintains an accurate count of wins for each player across multiple rounds, updating in real-time as games conclude. The round tracking system not only keeps count but also manages the progression of the match, clearly indicating to players their current position in the best-of-three sequence. When a match concludes (either through a player winning two games or through a decisive third game), the system handles the transition gracefully, offering options for a new match while preserving the integrity of the completed match's results. The reset mechanism for starting new best-of-three matches is carefully designed to clear all relevant state variables while maintaining the connection between players.

Synchronization System Benefits

The overall synchronization architecture provides several critical advantages for multiplayer gameplay. Move operations are handled atomically, meaning they either complete fully or not at all - there's no possibility of a move being partially applied or corrupted during transmission. This atomicity extends to all game state changes, including turn switches, win declarations, and score updates. The system maintains consistency across clients through a server-authoritative model, where all state changes must be validated and broadcast by the server before being applied locally by clients. This prevents state divergence even in challenging network conditions.The system's handling of network issues is particularly sophisticated. It includes mechanisms for:

- Detecting and handling client disconnects without disrupting the remaining player's experience
- Buffering and queuing messages to handle network latency spikes
- Maintaining game state integrity during temporary connection issues
- Providing clear feedback to players when network problems occur
- Allowing graceful recovery from connection interruptions
- This comprehensive approach ensures that players experience smooth, consistent gameplay even when network conditions are less than ideal, making the game suitable for both local network and internet play.

CONCLUSION:

During the development of our enhanced Tic-Tac-Tac game, our team faced a wide range of technical and collaborative challenges that pushed us to grow as developers and as effective team members. At the beginning of the project, many of us were unaware of collaboratively using GitHub, especially in a multibranch environment. Figuring out how to coordinate code changes, resolve merge conflicts, and keep a shared codebase stable was a learning curve, but turned into a huge advantage. We quickly learned that clear communication and planning were essential. Hosting meetings, exchanging ideas, and dividing tasks helped us align a shared vision and stay organized as the project became more complex. Working alongside dedicated and motivated colleagues made the experience productive and enjoyable, and encouraged a strong sense of teamwork.

One of the biggest technical challenges we addressed was implementing Best of 3 gameplay. Unlike a single round, this required us to persist and manage game state across multiple matches, synchronize wins between clients, and ensure the game loop continued to run smoothly without restarting the entire app. During this process, we had to carefully debug issues related to signal handling and flow control, especially between rounds, which ultimately improved our understanding of the game logic in real time. It was an excellent exercise in state management, event coordination, and client-server communication.

Another significant achievement was the addition of networked multiplayer functionality. We designed a system where two game instances could connect

and communicate via sockets, exchanging moves and game state in real time. This laid the groundwork for expansion into more social and interactive features, such as the conceptual friends system. Our idea was to allow users to log in with a username, search for other users, and maintain a visible friends list that updates based on their online presence. While this feature is still in its early stages, the underlying structure of friendserver.py, player1.py, and player2.py establishes a scalable model for future enhancements, such as friend requests, invites, and private matches.

Some of our key accomplishments throughout the project include the successful development of the "Best of 3" mode, which required cross-client coordination and constant monitoring of match results. We also improved the game flow, such as enabling clean client exits without crashes or hanging threads. A highlight of the project was the integration of single-player and multiplayer modes into a unified experience, allowing testers to easily navigate and evaluate both styles of play. Our branch-based development made it easy to experiment with features in isolation and then merge stable releases into the main branch.

Several potential areas for improvement remain, which we are excited about. Finalizing the friends system with persistent storage would allow players to build and maintain relationships over time. We're also interested in implementing real-time chat or notifications so players can interact more naturally during or between matches. A graphical lobby system could further enhance the user experience by providing an intuitive way to join, invite, or spectate matches with friends.

From a learning perspective, this project taught us a tremendous amount. We learned to value the importance of careful debugging, especially when working with asynchronous or networked systems, where issues can be subtle and time-related. We also gained a much stronger understanding of version control workflows, including how to avoid disrupting progress through responsible branching and pull requests. Most importantly, we developed as teammates: we learned to divide responsibilities, support each other's work, and solve problems collectively under pressure. These soft skills were just as valuable as the technical ones and will benefit us in future group projects and professional settings.

In conclusion, this project was an incredibly rewarding and educational experience. Not only did it push us to apply what we learned in a real-world programming environment, but it also allowed us to explore creative extensions of a classic game. We're proud of what we accomplished as a team, and the foundation we've built opens the door to even more ambitious features in the future.

REFERENCES:

[1] Jennings, N. (2024, December 7). *Socket Programming in Python (Guide)*. Real Python. https://realpython.com/python-sockets/

[2] B. S. K. Reddy, D. Singh, C. P. Agarwal, E. Mithil, S. Utukuru and C. R. Reddy, "Python based Advanced Multiplayer Gaming Environment," *2024 8th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, Kirtipur, Nepal, 2024, pp. 1189-1195, doi: 10.1109/I-SMAC61858.2024.10714784.