Tic-Tac-Toe

A Real-Time Multiplayer Game over TCP/IP

Course: CS5700 Computer Networking
Team members: Dixuan Zhao, Minyi Zhu, Yuanyuan Wu
Github: https://github.com/myz0601/cs5700-tictactoe

1. Introduction

The goal of this project is to design and implement a real-time multiplayer Tic-Tac-Toe game using a client–server architecture over TCP/IP. The system allows two players to connect to a centralized server, register their usernames, be matched into a session, and play a fully synchronized turn-based game. To demonstrate core networking concepts from the course, the project incorporates socket programming, I/O multiplexing with select(), multithreaded clients, a text-based application protocol, and persistent state tracking.

Beyond the required command-line client, we additionally implemented a Tkinter-based graphical user interface (GUI) and a Flask-powered web dashboard that displays player statistics. These extensions enhance usability while preserving the original networking architecture and protocol. This report presents an overview of the system features, design decisions, implementation methodology, testing results, and potential improvements for future iterations.

2. System Features

- Player Matching
  The server accepts connections, registers usernames, and pairs two players into a game session.
- Turn-Based Gameplay
  The server enforces turn order and validates moves, ensuring both clients always see a synchronized board.
- Real-Time Communication
  Clients support in-game chat, and all board updates and messages are delivered instantly over TCP.
- Error and Disconnect Handling
  Invalid moves, early quits, and unexpected disconnections are detected and handled gracefully.
- Persistent Statistics
  Wins, losses, and draws are saved in a JSON file and updated after each game.
- Multiple Client Interfaces

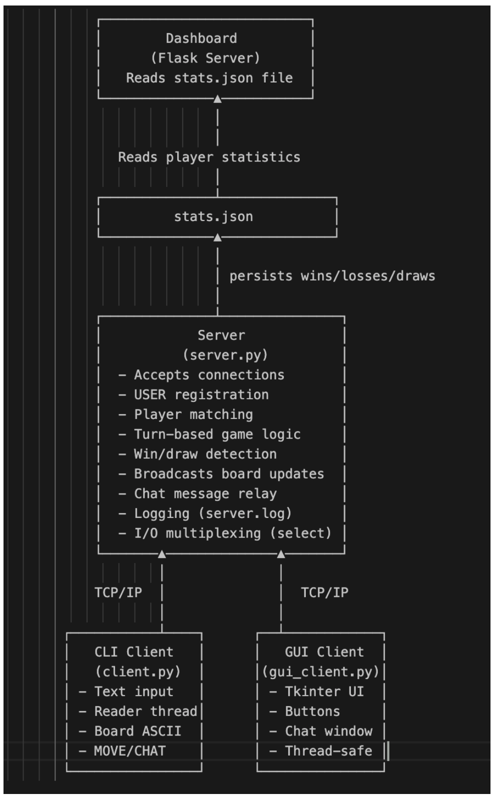
The system provides both a command-line client and an optional Tkinter GUI client.

- Web Dashboard (Optional)
  A simple Flask dashboard displays player statistics in a leaderboard format.

3. System Analysis and Design

The system architecture is composed of four major components:
(1) a centralized server responsible for game coordination;
(2) a command-line client;
(3) a graphical Tkinter client; and
(4) a Flask dashboard used for visualizing persistent statistics.

The following ASCII diagram illustrates the overall architecture:



```
            ┌──────────────────────────┐
            │        Dashboard         │
            │      (Flask Server)      │
            │   Reads stats.json file  │
            └─────────────▲────────────┘
                          │
                          │
               Reads player statistics
                          │
            ┌─────────────────────────────┐
            │          stats.json         │
            └─────────────▲───────────────┘
                          │
                          │ persists wins/losses/draws
                          │
            ┌───────────────────────────────┐
            │             Server            │
            │          (server.py)          │
            │  - Accepts connections        │
            │  - USER registration          │
            │  - Player matching            │
            │  - Turn-based game logic      │
            │  - Win/draw detection         │
            │  - Broadcasts board updates   │
            │  - Chat message relay         │
            │  - Logging (server.log)       │
            │  - I/O multiplexing (select)  │
            └───────▲───────────────▲───────┘
                    │               │
           TCP/IP   │               │  TCP/IP
                    │               │
        ┌────────────────┐     ┌────────────────┐
        │   CLI Client   │     │   GUI Client   │
        │   (client.py)  │     │(gui_client.py) │
        │ - Text input   │     │ - Tkinter UI   │
        │ - Reader thread│     │ - Buttons      │
        │ - Board ASCII  │     │ - Chat window  │
        │ - MOVE/CHAT    │     │ - Thread-safe  │
        └────────────────┘     └────────────────┘
```

4. Implementation Details and Methodology

4.1 Server (server.py)

The server manages game state and coordinates two players using TCP sockets.

Connection Flow: The server binds to port 5500, accepts two connections sequentially, and requires each player to register with USER <username> before starting the game.

I/O Multiplexing: During gameplay, select.select() monitors both player sockets simultaneously, allowing the server to handle chat messages from either player while enforcing turn order for moves.

Game Logic: The board is stored as a 9-element list. The check_winner() function examines 8 winning lines (3 rows, 3 columns, 2 diagonals) after each move. Statistics are persisted to stats.json when games end.

Protocol Commands:

MOVE row col — Server validates turn and cell availability, updates board, checks for win/draw

CHAT message — Broadcast to both players

QUIT — Opponent wins by forfeit

4.2 CLI Client (client.py)

The client uses two threads: the main thread handles user input, while a background reader thread receives and displays server messages. This keeps the UI responsive during gameplay.

Messages are framed with newlines. The recv_line() function reads byte-by-byte until \n, implementing application-layer framing over TCP's byte stream.

4.3 GUI Client (gui_client.py)

The Tkinter GUI uses a thread-safe queue to pass messages from the network thread to the UI thread. The main loop polls the queue every 100ms via root.after().

The 3×3 board consists of clickable buttons. Clicking sends a MOVE command to the server; the display updates only after receiving the server's authoritative BOARD response.

4.4 Dashboard (dashboard.py)

A Flask application that reads stats.json and server.log on each request, displaying a leaderboard table with player statistics.

5. Testing and User Manual

5.1 Testing

Test Case	**Result**

Normal gameplay (two players complete a game)	✓ Pass
Invalid move (occupied cell)	Error message returned
Out-of-turn move	"Not your turn" message
Chat during gameplay	Messages appear on both clients
Player disconnects	Opponent wins by default
CLI vs GUI client	Protocol compatible

5.2 User Manual

Start Server:

```
python server.py
```

Start Clients:

```
python client.py localhost 5500 Alice      # CLI
python gui_client.py localhost 5500 Bob    # GUI
```

CLI Commands:

- move r c — Place mark at row r, col c (0-2)
- chat message — Send chat message
- quit — Forfeit and exit

Board Positions:

```
(0,0)|(0,1)|(0,2)
-----+-----+-----
(1,0)|(1,1)|(1,2)
-----+-----+-----
(2,0)|(2,1)|(2,2)
```

Dashboard (Optional):

```
pip install flask
python dashboard.py
# Visit http://127.0.0.1:5000
```

Running on different machines (LAN testing):

Start the server on Machine A:

```
python server.py
```

On Machine A, find its local IP address (for example 192.168.1.23), and use this IP instead of localhost.

Start a client on Machine A:

```
python client.py 192.168.1.23 5500 Alice
```

Start another client on Machine B:

```
python client.py 192.168.1.23 5500 Bob
python gui_client.py 192.168.1.23 5500 Bob
```

As long as both machines are on the same network and the port is not blocked by a firewall, they can play against each other through the same server.

6. Improvements and Future Work

Although the current implementation satisfies the core requirements of the assignment, there are several directions in which the system can be improved and extended in future work.

a. Support for multiple concurrent games

At the moment, the server manages a single game between exactly two clients. A natural extension would be to support multiple ongoing games in parallel. Future work can focus on these improvements:

- When a new client connects, they can either create a room or join an existing one.
- The server would maintain a mapping from room IDs to game state and sockets.
- Each room runs a separate game loop, either in a dedicated thread or via an event-driven framework.

This would better reflect real-world networked services that handle many sessions concurrently, and would require more careful design for connection management and resource cleanup.

b. Improved robustness and reconnection

Currently, if one player disconnects, the server immediately declares the other player the winner and closes the sockets. Future versions could handle transient failures more gracefully:

- Allow a player to reconnect within a short timeout window using the same username and resume the existing game state.
- Distinguish between a user using “QUIT” and an unexpected network failure, and provide different user feedback.

These changes would make the system more fault-tolerant and closer to real online games.

c. Enhanced security and authentication

The current protocol and server are designed for a trusted environment and assume honest clients. Future improvements could include:

- A simple authentication mechanism, e.g., username + password or tokens, so that the statistics in stats.json cannot be easily spoofed.
- Basic input validation to mitigate impropriate messages.

Adding these features would connect the project more directly to topics such as secure communication and protocol hardening.

d. Richer front ends: web and mobile

Besides the CLI client and the Tkinter GUI client, the project can be extended with additional user interfaces:

- A browser-based client using HTML/JavaScript and WebSockets that speaks the same text-based protocol. The Python server could be extended or wrapped with a WebSocket layer, so users can play directly from a web page.
- Better styling and layout in the Tkinter GUI (e.g., highlighting the current player, disabling buttons for invalid cells, and clearer status messages).
- Mobile-friendly UI designs that could be used on tablets or phones.

These front-end improvements would not change the core networking logic but would demonstrate how the same protocol can support multiple types of clients.

e. Extended dashboard and analytics

The current stats and logging features (stats.json, server.log, and the optional Flask dashboard.py) can be expanded into a more capable monitoring system:

- Add more detailed metrics: total games played, average game duration, most active players, win streaks, etc.
- Provide live updates on the web dashboard (e.g., via WebSockets) instead of only static reads from stats.json.
- Visualize connection events and results over time, which can help debug and analyze system behavior.

This would turn the simple leaderboard into a small network monitoring and analytics tool.

f. Tournament mode and advanced game modes

Finally, on top of the basic Tic-Tac-Toe game, the networking framework could support more advanced features:

- A tournament mode where multiple games are scheduled and players progress through rounds.
- Alternative board sizes or game rules, implemented while keeping the same underlying protocol structure.
- Support for spectators, where additional clients connect in read-only mode and receive BOARD, INFO, and MSG updates but cannot send MOVE.

These extensions would push the design further toward a full multi-user networked application and provide additional opportunities to explore scalability and protocol design.

Overall, the current project provides a solid foundation: a working TCP-based client–server game with synchronized state, chat, statistics, and both CLI and GUI front ends. The improvements above outline how the same architecture can be evolved toward a more realistic, scalable, and secure networked system.

7. Conclusion

In this project, we designed and implemented a real-time multiplayer Tic-Tac-Toe game as a TCP/IP networking application. A Python server maintains the authoritative game state, coordinates two clients, and uses a simple text-based protocol to synchronize the board, turns, chat messages, and final results. Both the command-line client and the GUI client connect to the same server using TCP sockets, and a small JSON-based statistics module keeps track of wins, losses, and draws for each username. Logging on the server side records connection events and game outcomes, which helps us monitor and debug the system.

Overall, the project meets all of the requirements for the assignment: players can join, wait for an opponent, play turn-based games in real time, exchange chat messages, and see a consistent board on both sides. More importantly, the project gave us a concrete example of how a client server architecture, custom application-layer protocols, and basic concurrency techniques (threads and select) can be combined to build an interactive networked application. Although the game logic itself is simple, integrating networking, synchronization, and multiple front ends turned it into a complete and realistic networking project.

8. Reflection

Working on this project helped us move from theoretical concepts in the networking course to a working system that we could actually play with. At the beginning, even simple things like deciding where to store the board and how to format messages between endpoints were not obvious. We learned that keeping the server as the single source of truth and using a clear, line-based protocol made the design much easier to reason about and debug. Implementing “send_line, recv_line” also showed us why TCP is a byte stream without message boundaries, and why application-layer framing is necessary.

We also gained practical experience with concurrency and asynchronous events. On the server, select() allows us to handle chat and quit messages from both players at any time, while still enforcing turn order for moves. On the client side, the background reader thread keeps the UI responsive so the user can type or click while messages are arriving. Building both a CLI client and a GUI client taught us how the same protocol can support different types of front end, and how networking logic can be kept separate from presentation logic. If we had more time, we would like to extend the server to support multiple games in parallel and to add a WebSocket-based browser client. Overall, this project made the ideas of sockets, protocols, and shared state much more concrete and gave us more confidence in building networked applications in the future.