TCP-Based Multi-Session Networked Chess Application

Course: CS 348: Computer Networks

Team Members:

- Pranav Kumar Pandey (CS22BT043)
- Parikshit Gehlaut (CS22BT066)
- Amit Singh (CS22BT004)
- Mahesh Krishnam (MC22BT011)
- Harsh Jain (CE22BT003)

---

1. Introduction & Motivation

This report presents a networked multi-client chess application developed as a practical study in computer network programming principles using Python. The project's core motivation was to design and implement a system demonstrating proficiency in fundamental networking concepts applied to a real-time, interactive application. Key areas explored include:

- **Transport Layer Protocols:** Utilizing TCP (Transmission Control Protocol) for reliable, connection-oriented communication between clients and the server, ensuring ordered delivery of critical game moves and state updates.
- **Socket Programming:** Direct application of the BSD socket API via Python's socket module to establish and manage network connections (binding, listening, accepting, connecting, sending, receiving).
- **Client-Server Architecture:** Implementing a classical client-server model where a central server manages game state and logic, while multiple clients connect to interact with the system.
- **Network Concurrency:** Addressing the challenge of handling multiple simultaneous client connections on the server using multi-threading (threading module) and ensuring thread-safe access to shared resources through locking mechanisms (threading.Lock).
- **Application Layer Protocol Design:** Defining a simple protocol for client-server interaction, including message types (move, chat) and implementing message framing (length-prefixing using struct) to ensure reliable data unit reception over TCP streams.
- **Data Serialization for Network Transmission:** Employing pickle to serialize Python game state objects into byte streams suitable for transmission across the network.

The project aims to build a functional networked application that tackles common challenges in network programming, such as state synchronization, real-time updates, concurrency, and reliable data transport.

---

2. Project Goals

The primary objectives from a networking perspective were:

- **Establish Reliable Communication Channels:** Utilize TCP sockets to create persistent, reliable connections between clients and the server.
- **Implement Server-Side Multiplexing:** Design the server to handle connections from multiple clients concurrently, multiplexing independent game sessions over these connections.
- **Transmit Game State and Actions over TCP/IP:** Define and implement mechanisms to reliably send chess moves, chat messages, and complete game state updates across the network.
- **Ensure Data Integrity during Transmission:** Implement an application-level message framing protocol (length-prefixing) on top of TCP to guarantee that clients receive complete and uncorrupted data units (serialized game states/messages), mitigating issues arising from TCP's stream-based nature.
- **Achieve Real-time State Synchronization:** Broadcast game state updates efficiently to relevant clients to maintain a synchronized view of the game across the network with minimal perceived latency.
- **Manage Network Sessions:** Handle the lifecycle of client connections, including pairing, game association, and graceful disconnection detection and cleanup.
- **Server-Authoritative State and Time:** Ensure the server is the single source of truth for game state and player timers, synchronizing this information across connected clients.

3. System Architecture & Network Design (Enhanced Detail)

The application implements a **multi-session, concurrent Client-Server** architecture over **TCP/IP**, designed to handle multiple independent chess games simultaneously.

- **Transport Protocol (TCP):** Chosen for its inherent reliability and ordered stream delivery, crucial for ensuring chess moves, state updates, and chat messages arrive correctly and in sequence. socket.SOCK_STREAM is used throughout.
- **Server Core:** The central Python script (server.py) acts as the network endpoint and game orchestrator.

    - **Connection Handling (Thread-per-Client):** The main server thread listens for incoming TCP connections using socket.listen() and socket.accept(). Upon accepting a connection, it immediately spawns a dedicated **handler thread** (threading.Thread targeting handle_client) for that specific client socket. This **thread-per-client** model isolates network I/O for each client, preventing one slow or unresponsive client from blocking others or the main accept loop.
    - **Player Pairing & Game Instantiation:**
        - A **waiting queue** (waiting_players dictionary) holds connections awaiting an opponent.
        - A pairing_lock ensures thread-safe access to this queue and the client_to_game mapping.
        - When a client handler thread starts, it acquires the pairing_lock. If the queue is not empty, it pops a waiting client, generates a unique game_id (using uuid.uuid4), and instantiates a Game object. If the queue is empty, the current client is added to the queue.
        - This strategy ensures that clients are paired efficiently as they connect.
    - **Multi-Game Management (Game Class & games Dictionary):**
        - The core of the multi-game support lies in the Game class. Each instance of this class encapsulates the *entire state* for a single chess match: piece lists and locations, captured pieces, turn step, player timers, chat history, game status (game_started, game_over, winner), and references to the two connected player sockets (players, player_conns).
        - Crucially, each Game instance has its own internal game_state_lock (threading.Lock) to protect its state variables during modifications (e.g., within apply_move, update_timers, add_chat_message), ensuring data integrity within a single game session even when accessed by different threads (client handlers or the periodic updater).
        - A global games dictionary stores all active Game instances, keyed by their game_id. A separate games_lock protects this dictionary during addition, removal, or iteration (e.g., by the periodic update thread).
    - **Message Routing:** When a client handler thread receives a message (move or chat), it first identifies the client's game_id from the shared client_to_game map (using pairing_lock for safe access). It then retrieves the corresponding Game instance from the games dictionary (using games_lock) and invokes the appropriate method (apply_move or add_chat_message) on *that specific instance*. This ensures actions affect only the correct game.
    - **Targeted Broadcasting:** State updates (broadcast_game_state) are not sent globally. The function retrieves the specific Game instance, gets the list of connections *only* for that game (game.get_player_connections()), and sends the updated state exclusively to those participants. This prevents clients from receiving irrelevant data from other games.
- **Client Core:** The client.py script connects to the server and interacts based on server messages.

    - **Connection & Assignment:** Connects via TCP (socket.connect). It waits to receive its assigned color ('white' or 'black') from the server, which indicates it has been paired into a game.
    - **Dedicated Receive Thread:** Similar to the server, the client uses a separate receive_updates thread to handle incoming network data (socket.recv). This prevents network blocking from freezing the main GUI thread.
    - **State Synchronization:** The receive thread uses the implemented framing protocol (receive_one_message) to get complete state updates. It then updates the shared game_state dictionary (protected by game_state_lock), which the main thread reads for rendering the GUI.
- **Application Layer Protocol & Framing:**
    - **Serialization:** pickle is used to convert Python dictionaries (representing actions or game state) into byte streams.
    - **Framing (Server -> Client):** The **length-prefix framing** strategy is implemented using struct. The server calculates the pickled message size, packs it into a 4-byte network-order integer (>I), and sends this header before the payload. This allows the client (receive_one_message) to reliably read the header, determine the exact payload size, and read the complete message before attempting to unpickle, thus avoiding errors from TCP stream artifacts.

4. Technology Stack

- **Core Networking:** Python socket module (BSD Sockets API)
- **Transport Layer:** TCP/IP (via socket.SOCK_STREAM)
- **Concurrency:** Python threading module & threading.Lock (for managing concurrent socket connections and shared state)
- **Message Framing:** Python struct module (for packing/unpacking length prefixes)
- **Data Serialization:** Python pickle module (for network payload preparation)
- **GUI / Presentation:** Pygame
- **Language:** Python 3.x

```
+-------------------------+      +-------------------------+      +-------------------------+
| Client 1 (Addr A:P1)    | ---- |                         | ---- | Client 3 (Addr C:P3)    |
| [Pygame, Socket Conn S1]|      |                         |      | [Pygame, Socket Conn S3]|
+-------------------------+      |      TCP/IP Network     |      +-------------------------+
          |                      |                         |                   |
          v (TCP Connect)        | (Connections S1,S2,S3,S4) |       (TCP Connect) v
+-------------------------------------------------------------------------------------------+
|                               Server (Python @ 0.0.0.0:PORT)                              |
| +---------------------+  +---------------------------+  +---------------------------------+ |
| | Listening Socket L1 |->| Accept -> New Client Thread |->| Pairing Logic (pairing_lock)    | |
| | (Bound, Listening)  |  | (Handles Socket S1, S2...)  |  | {S1:A:P1 wait},{S3:C:P3 wait} ->| |
| +---------------------+  +---------------------------+  +--+------------------------------+ |
|                                                          | Creates Game Instance          |
|           +--------------------------------------------+--------------------------------+ |
|           |             Game Manager (games_lock - {game_id: Game Instance})            | |
|           | +----------------------+   +----------------------+                         | |
|           | | Game A (S1(W), S2(B))|   | Game B (S3(W), S4(B))| ...                     | |
|           | | [State, Logic, Lock] |   | [State, Logic, Lock] |                         | |
|           | +----------------------+   +----------------------+                         | |
|           +-------------+-----------------------------+---------------------------------+ |
|                         | Sends Framed State Updates  | (Via Client Handler Threads)      |
|                         v (Targeted Broadcast)        v                                   |
| +-----------------------------------------------------------------------------------------+ |
| |                      Periodic Update Thread (Updates all Game Timers)                  | |
| +-----------------------------------------------------------------------------------------+ |
+-------------------------------------------------------------------------------------------+
          ^ (TCP Connect)        |                         |       (TCP Connect) ^
          |                      |                         |                   |
+-------------------------+      |                         |      +-------------------------+
| Client 2 (Addr B:P2)    | ---- |                         | ---- | Client 4 (Addr D:P4)    |
| [Pygame, Socket Conn S2]|      |                         |      | [Pygame, Socket Conn S4]|
+-------------------------+      +-------------------------+      +-------------------------+
```

5. Features Implemented (Detailed Breakdown)

This project implements several key features, leveraging specific networking and programming techniques:

- **Networked Gameplay & Client-Server Model:** Establishes reliable TCP connections (socket) for real-time chess matches. The server acts as the authoritative source for all game logic and state.
- **Multi-Client / Multi-Game Architecture:**

    - **Strategy:** Implemented using a combination of thread-per-client connection handling, a central games dictionary protected by games_lock, and a dedicated Game class for state encapsulation. Player pairing occurs via a waiting queue protected by pairing_lock.
    - **Details:** The server can host multiple independent 2-player chess sessions concurrently. Each Game instance maintains its own board state, timers, and chat history, completely isolated from other games. Client messages are routed by the server to the correct Game instance based on the client_to_game mapping. State broadcasts are targeted only to the participants of the specific game being updated.
- **Graphical User Interface (Pygame):** Provides an interactive visual representation of the game elements (board, pieces, valid moves, check status, chat, timers) and handles user input (mouse clicks, keyboard).
- **Core Chess Logic & State Management:**
    - **Strategy:** Game rules (movement, capture, basic promotion) are implemented in helper functions (check_options, check_pawn, etc.) and applied within the Game.apply_move method. Game state is encapsulated within the Game object.
    - **Details:** Includes validation for standard piece moves. Captures remove opponent pieces and add them to the capturing player's list. Pawns reaching the opposite rank are automatically promoted to Queens. The turn_step variable dictates the current turn.
- **Time-Bounded Gameplay (Time Control):**
    - **Strategy:** Implemented using server-side timer management and periodic updates. Each Game instance tracks white_time, black_time, and last_timer_update. A dedicated background thread (periodic_updates) triggers timer decrements.
    - **Details:** Each player starts with INITIAL_TIME_SECONDS (e.g., 1200 seconds / 20 minutes). The server-side Game.update_timers method calculates elapsed time since the last_timer_update and subtracts it from the *current* player's remaining time. This calculation happens periodically (driven by the periodic_updates thread, roughly every BROADCAST_INTERVAL) and also implicitly right before a move is processed (as the elapsed time is calculated relative to the start of the turn). If a player's time (white_time or black_time) drops to zero or below, the game_over flag is set to True, and the winner is declared as the *opponent*. This state change is then propagated to clients via the next broadcast. Clients simply display the time values received from the server.
- **In-Game Chat:**
    - **Strategy:** Implemented by storing chat messages within each Game instance's chat_history list and broadcasting the updated game state (which includes the chat history) to the relevant players.
    - **Details:** Clients send chat messages (type 'chat') containing the text. The server's run_game_communication function receives this, finds the correct Game instance, and calls game.add_chat_message. This method appends a dictionary {sender, text, timestamp} to the game's chat_history list (limiting its size to

prevent unbounded growth). When the game state is broadcast via broadcast_game_state, the updated chat_history is included. The client's draw_chat function iterates through this received history, formats messages into bubbles based on the sender, handles word wrapping, and displays them in a scrollable panel. Chat is isolated per game session.

- **Concurrency & Synchronization:**
    - **Strategy:** Uses a thread-per-client model for network I/O, a separate thread for periodic tasks, and multiple threading.Lock objects (pairing_lock, games_lock, Game.game_state_lock) for mutual exclusion on shared resources.
    - **Details:** This prevents blocking on network operations and allows concurrent game processing. Locks ensure that critical shared data structures (waiting queue, game dictionary, individual game states) are not corrupted by simultaneous access from different threads.
- **Reliable Messaging (Framing):**
    - **Strategy:** Implemented using length-prefix framing for server-to-client messages via the struct module.
    - **Details:** The send_framed_message function on the server prefixes each pickled message with its 4-byte length. The client's receive_one_message function reads this length first, then reads exactly that number of bytes for the payload, ensuring complete messages are unpickled and preventing client freezes or errors due to partial reads from the TCP stream.
- **Data Serialization (pickle):**
    - **Strategy:** Python's pickle module is used to convert Python objects (dictionaries representing game state or actions) into byte streams for network transmission and back.
    - **Details:** This simplifies sending complex data structures but requires both client and server to be written in Python and can have security implications if used in untrusted environments (not a major concern for this project scope).
- **Win/Loss Conditions:** Handled based on King capture (within apply_move), time expiry (within update_timers), and opponent disconnection (within remove_player). The game_over and winner attributes in the Game state reflect the outcome.

6. Getting Started / How to Run

Prerequisites:

- Python 3.6 or higher installed.
- pip (Python package installer) available.

Installation:

Install the Pygame library: **- pip install pygame**

Setup:

1. Place the server.py and client.py files in the same directory.
2. Create a subdirectory named assets in that same directory.
3. Inside the assets directory, create another subdirectory named images.
4. Place all the required chess piece image files (e.g., white pawn.png, black king.png, etc.) inside the assets/images/ directory. Ensure filenames match those used in client.py.
   (Note: If running server and client on different machines, ensure the server machine is reachable, potentially adjusting firewall rules for the specified PORT.)

Running the Application:

1. **Start the Server:** Open a terminal or command prompt, navigate to the directory containing the files, and run:

```
python3 server.py
```

2. Start Clients: Open separate terminal windows for each client and run:

```
python3 client.py
```

3. Gameplay: Interact with the Pygame window as described previously.

---

7. Demonstration Link

A screen recording demonstrating the network setup, multiple clients connecting, independent gameplay/chat/timers across concurrent sessions, and disconnection handling can be found at:

[Youtube Video Link]

---

8. Conclusion and Future Work

This project successfully demonstrates the application of fundamental computer networking concepts (TCP, sockets, concurrency, framing) to build a real-time, multi-user application. The client-server architecture effectively centralizes game logic, while the multi-threaded server handles concurrent client connections. The implementation of length-prefix framing proved crucial for reliable state synchronization over TCP.

Potential Network-Related Future Enhancements:

- **Client-Side Framing:** Implement message framing for client-to-server communication for added robustness and consistency.
- **Application-Level Keepalives:** Implement periodic "ping/pong" messages to detect dead connections more proactively than relying solely on TCP timeouts or errors during send/recv.
- **Improved Disconnection Handling:** Implement more sophisticated logic for game state recovery or offering draws if a player temporarily disconnects and reconnects (requires session persistence).
- **Alternative Serialization:** Investigate more language-agnostic and potentially more efficient serialization formats like JSON (with custom object encoding) or Protocol Buffers, especially if planning for clients in different languages.
- **Network Security:** Implement TLS/SSL encryption over the socket connection to protect game data and chat messages from eavesdropping, particularly if deployed over the public internet.
- **UDP Exploration:** Explore using UDP for potentially less critical, high-frequency updates (like cursor positions if added) while keeping critical game state on TCP.
- **Scalability Analysis:** Profile server performance under load (many concurrent games) and investigate asynchronous frameworks (asyncio) as an alternative to threading for potentially better resource utilization with very high connection counts.