Network Programming

Ola Magdi Mortadi El-Shiekh

Dr.Elhusseny & Eng. Mostafa Atlam
FEE_CSE Dept 55.

The code provided consists of a server and a client application for a Connect 4 game, implemented in Python using sockets for network communication and tkinter for the client GUI. Here's a detailed breakdown of the concept and implementation of sockets in this context, as well as an overview of how the game logic and GUI are handled.

Concept of Sockets

Sockets provide a way to establish a communication channel between different processes, either on the same machine or over a network. In this case, sockets are used to enable real-time communication between the game server (which manages the game state and logic) and multiple client programs (which provide a graphical interface for the players).

Server Code Breakdown

1. Socket Initialization:

- socket.socket(socket.AF_INET, socket.SOCK_STREAM): This line creates a new socket using IPv4 addressing (AF_INET) and TCP (SOCK_STREAM), which is reliable and connection-oriented, suitable for a game where moves must be reliably transmitted in order.

2. Binding and Listening:

- server_socket.bind((host, port)) and server_socket.listen(2): These lines bind the socket to a local address (127.0.0.1 on port 5000) and start listening for incoming connections, with a backlog of 2 (it can queue up to 2 connection requests).

3. Accepting Connections:

- conn, addr = server_socket.accept(): This line blocks until a new client connects, returning a new socket object (conn) that is used to communicate with the client, and the address (addr) of the client.

4. Handling Client Requests in Threads:

- Each client connection is handled in a separate thread (threading.Thread(target=client_thread, args=(conn, addr, board, current_player)).start()). This allows the server to manage multiple clients simultaneously without one blocking the others.

Client Code Breakdown

1. Socket Initialization and Connection:

- Similar to the server, the client creates a socket and connects to the server using self.server_socket.connect(('127.0.0.1', 5000)). This establishes a TCP connection to the server's listening socket.

2. Sending Moves to Server:

- self.server_socket.send(str(col).encode()): When a player clicks a column in the GUI, this line sends the column number as a move to the server. The move is encoded as bytes, which is necessary for socket communication.

3. Receiving and Processing Server Responses:

- response = self.server_socket.recv(1024).decode(): The client waits to receive a response from the server, which includes updates about the game state or notifications about game events (e.g., a player winning). The response is decoded from bytes to a string.

4. Updating GUI Based on Server Response:

- The update_board method updates the GUI based on the server's response, changing the color of the buttons to reflect the moves made by the players.

Explanation of Game Play

- Players take turns making moves through the GUI.
- Each move is sent to the server, which updates the game state, checks for a win, and then sends back information about the move to all clients.
- The clients receive this information and update their GUIs accordingly.

This architecture ensures that the game logic remains centralized on the server, providing a consistent view of the game state to all clients, and reducing the complexity of the client code. The use of threads on the server allows handling each client interaction independently, providing a responsive gaming experience to multiple players.