CS421

Programming Assignment 2

26.02.2023

Gökberk Keskinkılıç

21801666

Implementing a Tic-Tac-Toe Game over TCP

In this project a simple game was implemented over a TCP connection. Libraries used are sys, socket and threading. Client is only responsible for delivering the input in correct form. Server handles all the work, as it should. This provides safety to the application. Error messages and input handling is done at the server side.

Here is a sample output from server console throughout the application. It prints desired messages to the console and briefly displays moves of the clients. At the end it print final board state and the winning side, later shuts down the server.

```
gokiberk@GokiBook-Pro code % python3 TicTacToeServer.py 8081
Server runs...
A client is connected, and it is assigned with the symbol X and ID=0.
A client is connected, and it is assigned with the symbol O and ID=1.
The game has started.
Received X on (1,4). It is an illegal move.
Received X on (0,0). It is a legal move.
Received O on (2,2). It is a legal move.
Received X on (2,1). It is a legal move.
Received O on (1,1). It is a legal move.
Received X on (2,1). It is an illegal move.
Received X on (0,2). It is a legal move.
Received O on (2,0). It is a legal move.
Received X on (0,1). It is a legal move.

Final Board State:
X|X|X
_|O|_
O|X|O

Player 0 (X) has won the game!
Connections to clients closed.
Server stopped.
gokiberk@GokiBook-Pro code %
```

Fig.1 Console of the server side.

These are sample outputs from Client0 Client1. Since Client0 it is the first one to connect, it makes the first movement and has the symbol X. Error messages are handled at the server side.

```
gokiberk@GokiBook-Pro code % python TicTacToeClient.py 8081
Connected to the server.
Your symbol is X and your ID is 0.

Board State:
_|_|_
_|_|_
_|_|_

Your turn!
Put your mark to (row, col): aa
Invalid input format. Please enter row and column in the format
'row,col'. i.e. 1,1
Put your mark to (row, col): 1,4
Invalid move. Try again.

Board State:
_|_|_
_|_|_
_|_|_

Your turn!
Put your mark to (row, col): 0,0

Board State:
X|_|_
_|_|_
_|_|_

Waiting for Player 1's move.
```

Fig.2 Console of Client0

```
Waiting for Player 0's move.

Board State:
X|_|X
_|O|_
_|X|O

Your turn!
Put your mark to (row, col): 2,0

Board State:
X|_|X
_|O|_
O|X|O

Waiting for Player 0's move.

Final Board State:
X|X|X
_|O|_
O|X|O

Player 0 (X) has won the game!
The game has ended.
```

Fig.3 Console of Client1

```python
import socket
import sys

class TicTacToeClient:
    def __init__(self, port):
        self.client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client.connect(('localhost', port))

    def listen_server(self):
        while True:
            data = self.client.recv(1024)
            if data:
                message = data.decode()
                #if is taken only at the first message for symbol and client_id
                if "#" in message:
                    symbol, client_id = message.split('#')
                    print(f"Connected to the server.\nYour symbol is {symbol} and your ID is {client_id}.")
                else:
                    print(message)
                    if "Your turn!" in message:
                        self.make_move()

    def make_move(self):
        while True:
            try:
                move = input("Put your mark to (row, col): ")
                row, col = map(int, move.split(","))
                self.client.send(move.encode())
                break
            except ValueError:
                print("Invalid input format. Please enter row and column in the format 'row,col'. i.e. 1,1")


if __name__ == "__main__":
    port = int(sys.argv[1])
    client = TicTacToeClient(port)
    client.listen_server()
```

Fig.4 Implementation of the client

Client has two simple methods, it provides functionality to listen to server and to deliver the move to the server after parsing. In listen_server method, "`if "#" in message`" statement allows client to understand the first message from the handle_client method. That method delivers the message in the form of "`conn.send(f"{symbol}#{player_id}".encode())`" which uses pound sign as a separator. This is the only message that uses pound sign, so it is not possible to cause any trouble. In make_move method, there is a use of try catch block to implement error handling. When client sends the input without a comma, client handles this error. Further mechanisms are handled at the server side.

Server side starts with assigning ids and symbols to the clients that connects to the server. This is done in accept_connections method. While there are not 2 clients, server awaits to accept connection. Once there is a request it accepts and assigns a thread to the client. When there are 2 clients connected, it executes the start_game method.

```
def accept_connections(self):
    print("Server runs...")
    while len(self.clients) < 2:
        conn, addr = self.server.accept()
        self.clients.append((conn, addr))
        threading.Thread(target=self.handle_client, args=(conn, addr)).start()

    print("The game has started.")
    self.start_game()
```

Fig.5 accept_connections method

Here is the main driver method of the application, start_game. It calls required helper functions to send board state to clients, and handle their turns until the game is not over or the board is full.

```
def start_game(self):
    while True:
        self.send_board_state()
        self.handle_turn()
        if self.is_game_over():
            winner = (self.turn + 1) % 2  # the player who made the last move is the winner
            print(f"\nFinal Board State:\n{self.board_to_string()}")
            print(f"Player {winner} ({self.symbols[winner]}) has won the game!")
            self.send_game_over_message(winner, True)
            break
        elif not any('_' in row for row in self.board):
            print(f"\nFinal Board State:\n{self.board_to_string()}")
            print("The game is a draw!")
            self.send_game_over_message(None, False)
            break
```

Fig.6 start_game method

Below send_board_state is invoked from start_game method. It indicates to clients whose turn to play. message_sent is assigned to True to print that statement only once on awaiting client's console. Without this statement, it prints repeatedly on the console.

```
def send_board_state(self):
    for i, (conn, addr) in enumerate(self.clients):
        if i == self.turn:
            message = f"\nBoard State:\n{self.board_to_string()}\nYour turn!"
            conn.send(message.encode())
        elif not self.message_sent[i]:
            message = f"\nBoard State:\n{self.board_to_string()}\nWaiting for Player
            conn.send(message.encode())
            self.message_sent[i] = True
```

Fig.7 send_board_state

Depending on the result winner is announced or the game is declared as a draw. Below is the handle_turn method, responsible for receiving the moves from the client. It assigns the row and column, then checks its validity. If it is valid, it updates the board array then prints the board.

```
def handle_turn(self):
    with self.turn_lock:
        conn, addr = self.clients[self.turn]
        move = conn.recv(1024).decode().strip().split(',')
        row, col = int(move[0]), int(move[1])

        if self.is_valid_move(row, col):
            self.board[row][col] = self.symbols[self.turn]
            self.print_move_message(self.turn, row, col, True)
            self.turn = (self.turn + 1) % 2
            self.message_sent = [False, False]  # Reset the message_sent
        else:
            conn.send("Invalid move. Try again.".encode())
            self.print_move_message(self.turn, row, col, False)
```

Fig.8 handle_turn method

is_valid_move checks the validity by looking at the index given is empty and within the boundary.

```
def is_valid_move(self, row, col):
    if 0 <= row < 3 and 0 <= col < 3 and self.board[row][col] == '_':
        return True
    return False
```

Fig.9 is_valid_move method

print_move_message is a simple method printing out to the server console movement details.

```
def print_move_message(self, player, row, col, is_legal):
    if is_legal:
        message = f"Received {self.symbols[player]} on ({row},{col}). It is a legal move."
    else:
        message = f"Received {self.symbols[player]} on ({row},{col}). It is an illegal move."
    print(message)
```

Fig.10 print_move_message method

is_game_over checks the status of the game by checking the elements in the board array.

```
def is_game_over(self):
    for row in self.board:
        if row[0] == row[1] == row[2] != '_':
            return True
    for col in range(3):
        if self.board[0][col] == self.board[1][col] == self.board[2][col] != '_':
            return True
    if self.board[0][0] == self.board[1][1] == self.board[2][2] != '_':
        return True
    if self.board[0][2] == self.board[1][1] == self.board[2][0] != '_':
        return True
    return False
```

Fig.11 is_game_over method

handle_client method prints on the server console when a client is connected.

```
def handle_client(self, conn, addr):
    player_id = len(self.clients) - 1
    symbol = self.symbols[player_id]
    conn.send(f"{symbol}#{player_id}".encode())
    print(f"A client is connected, and it is assigned with the symbol {symbol} and ID={player_id}.")
```

Fig.12 handle_client method

board_to_string method creates the tables in a nicer way on the console, it print “_” if the index is not used, else prints relevant symbol as it can be seen from Fig.2 and Fig.3. send_board_state prints out the board created by the board_to_string method to both clients.

In conclusion, server code creates a communication base for the clients and keeps the state of the board while checking the game status. Client side is only responsible for delivering the desired movement in correct format to server.