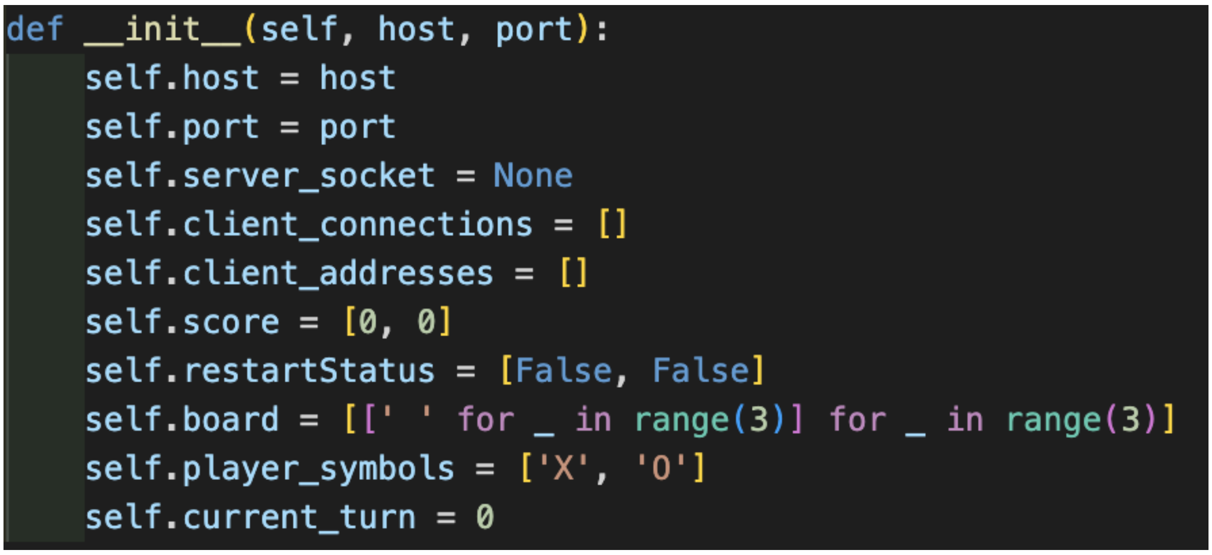
CS-421: Computer Networks
Mehmet Berk Türkçapar
21902570 - Section 1

Programming Assignment 2 - Tic-Tac-Toe Game

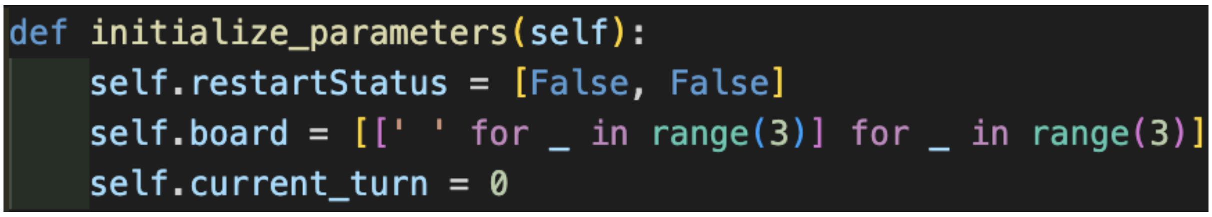
I used a class-based implementation for both the Client and the Server side. As an additional feature, I have implemented a scoreboard system and the option for the players to either replay or quit after a game ends. I have used three terminals while executing my code: one for running the server and two for running the clients.

Server Side Implementation



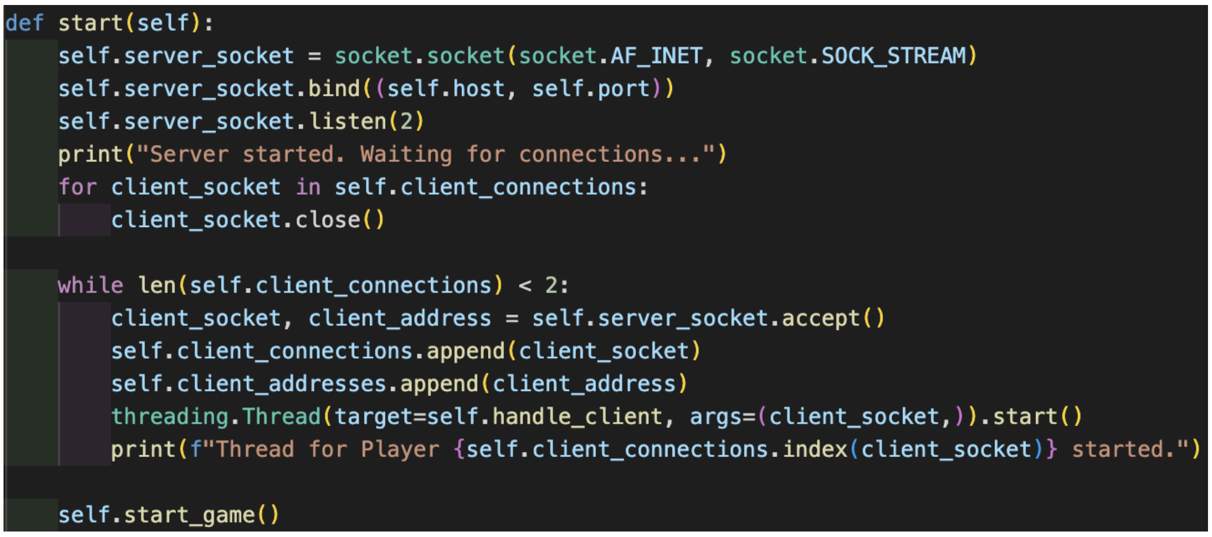
```
def __init__(self, host, port):
    self.host = host
    self.port = port
    self.server_socket = None
    self.client_connections = []
    self.client_addresses = []
    self.score = [0, 0]
    self.restartStatus = [False, False]
    self.board = [[' ' for _ in range(3)] for _ in range(3)]
    self.player_symbols = ['X', 'O']
    self.current_turn = 0
```

__init__(self, host, port): This is the constructor method for the server class. It initializes various attributes, such as the host and port for the server socket, the client connections and addresses, the game score, the game board, the player symbols ('X' and 'O'), the current turn.



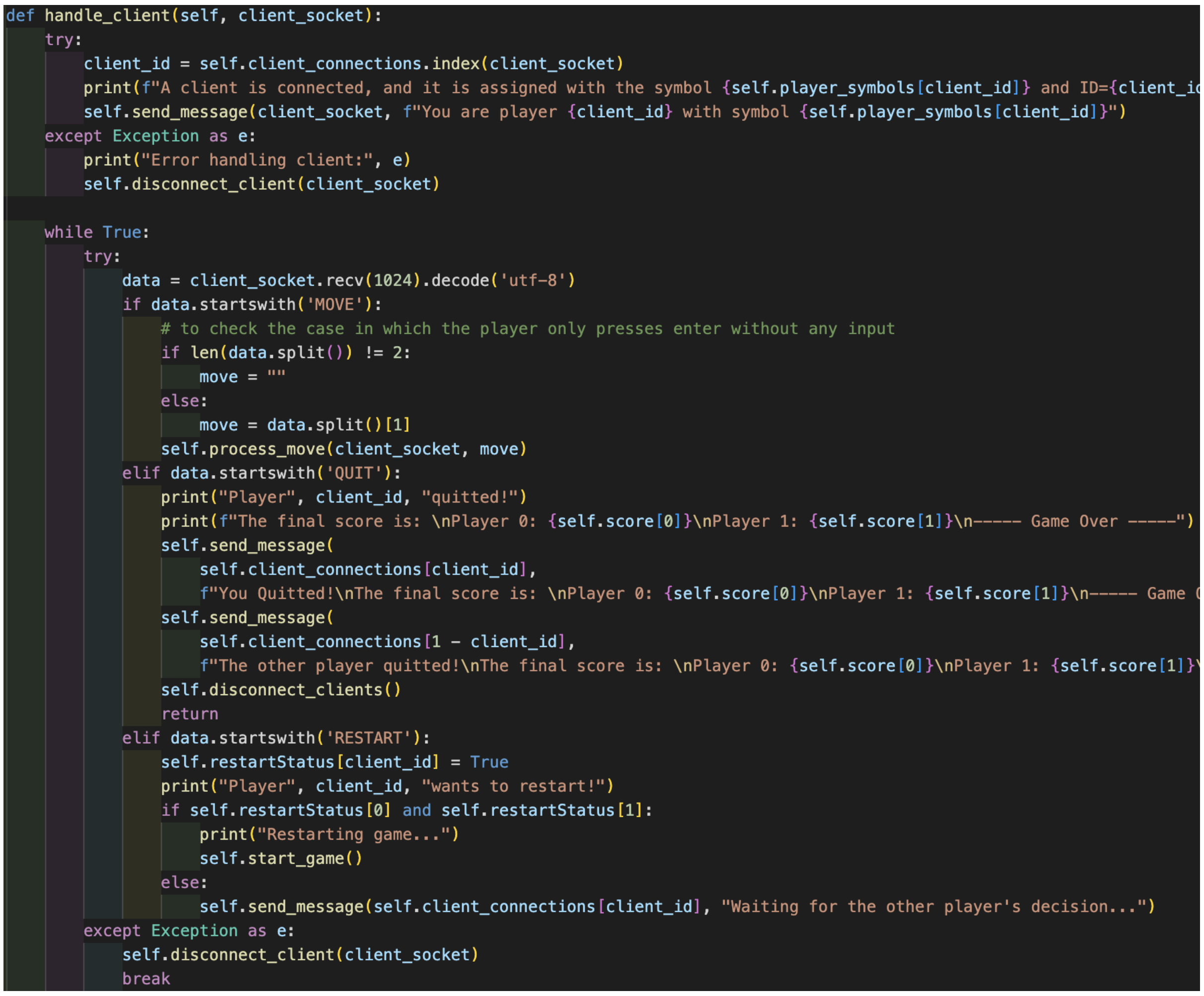
```
def initialize_parameters(self):
    self.restartStatus = [False, False]
    self.board = [[' ' for _ in range(3)] for _ in range(3)]
    self.current_turn = 0
```

initialize_parameters(self): This method is used to reset the game parameters to their initial values. It sets the restart status, clears the game board, and resets the current turn.



```python
def start(self):
    self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.server_socket.bind((self.host, self.port))
    self.server_socket.listen(2)
    print("Server started. Waiting for connections...")
    for client_socket in self.client_connections:
        client_socket.close()

    while len(self.client_connections) < 2:
        client_socket, client_address = self.server_socket.accept()
        self.client_connections.append(client_socket)
        self.client_addresses.append(client_address)
        threading.Thread(target=self.handle_client, args=(client_socket,)).start()
        print(f"Thread for Player {self.client_connections.index(client_socket)} started.")

    self.start_game()
```

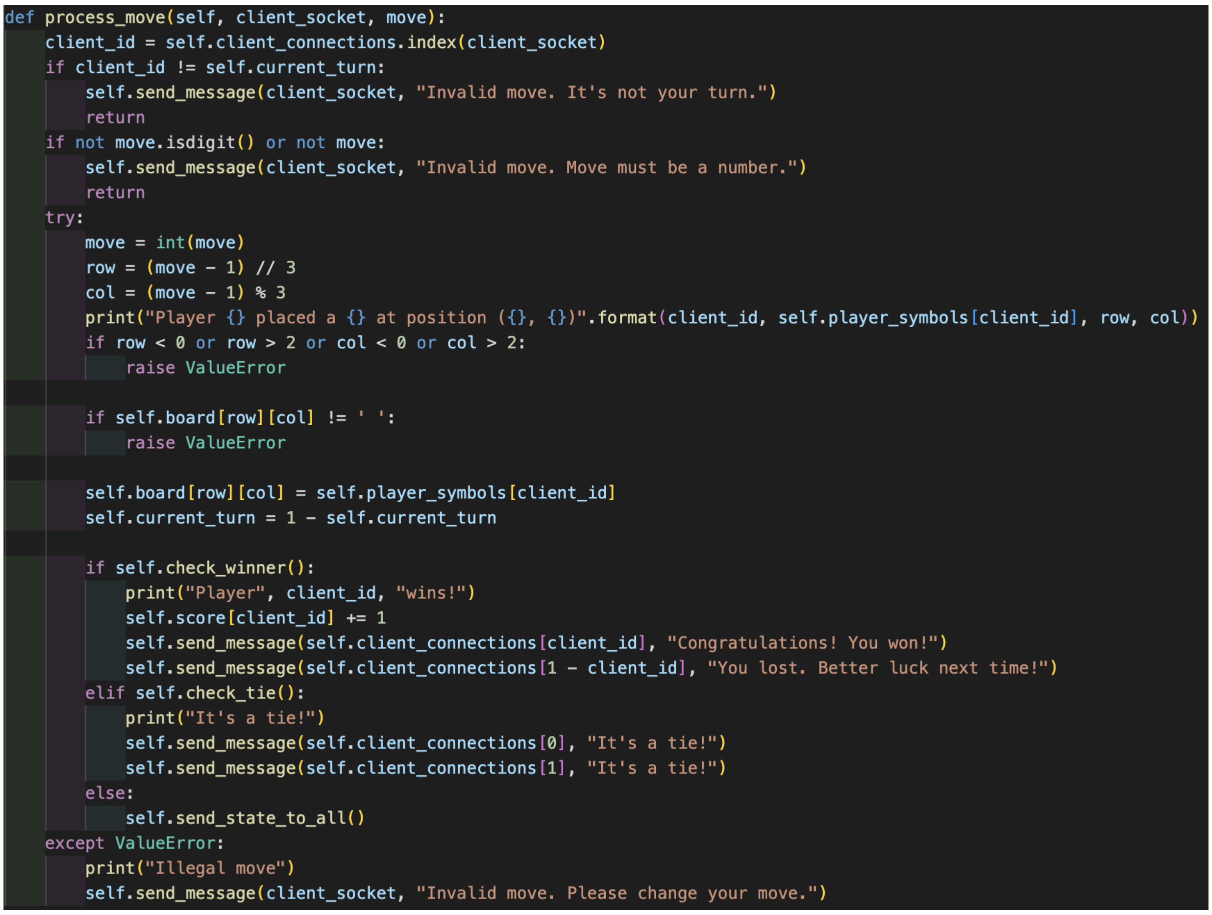
start(self): This method is the entry point for starting the server. It creates a server socket, binds it to the specified host and port, and listens for incoming connections. It accepts two client connections, starts a new thread for each client, and begins the game.



```python
def handle_client(self, client_socket):
    try:
        client_id = self.client_connections.index(client_socket)
        print(f"A client is connected, and it is assigned with the symbol {self.player_symbols[client_id]} and ID={client_ic
        self.send_message(client_socket, f"You are player {client_id} with symbol {self.player_symbols[client_id]}")
    except Exception as e:
        print("Error handling client:", e)
        self.disconnect_client(client_socket)

    while True:
        try:
            data = client_socket.recv(1024).decode('utf-8')
            if data.startswith('MOVE'):
                # to check the case in which the player only presses enter without any input
                if len(data.split()) != 2:
                    move = ""
                else:
                    move = data.split()[1]
                self.process_move(client_socket, move)
            elif data.startswith('QUIT'):
                print("Player", client_id, "quitted!")
                print(f"The final score is: \nPlayer 0: {self.score[0]}\nPlayer 1: {self.score[1]}\n----- Game Over -----")
                self.send_message(
                    self.client_connections[client_id],
                    f"You Quitted!\nThe final score is: \nPlayer 0: {self.score[0]}\nPlayer 1: {self.score[1]}\n----- Game (
                self.send_message(
                    self.client_connections[1 - client_id],
                    f"The other player quitted!\nThe final score is: \nPlayer 0: {self.score[0]}\nPlayer 1: {self.score[1]}\
                self.disconnect_clients()
                return
            elif data.startswith('RESTART'):
                self.restartStatus[client_id] = True
                print("Player", client_id, "wants to restart!")
                if self.restartStatus[0] and self.restartStatus[1]:
                    print("Restarting game...")
                    self.start_game()
                else:
                    self.send_message(self.client_connections[client_id], "Waiting for the other player's decision...")
        except Exception as e:
            self.disconnect_client(client_socket)
            break
```

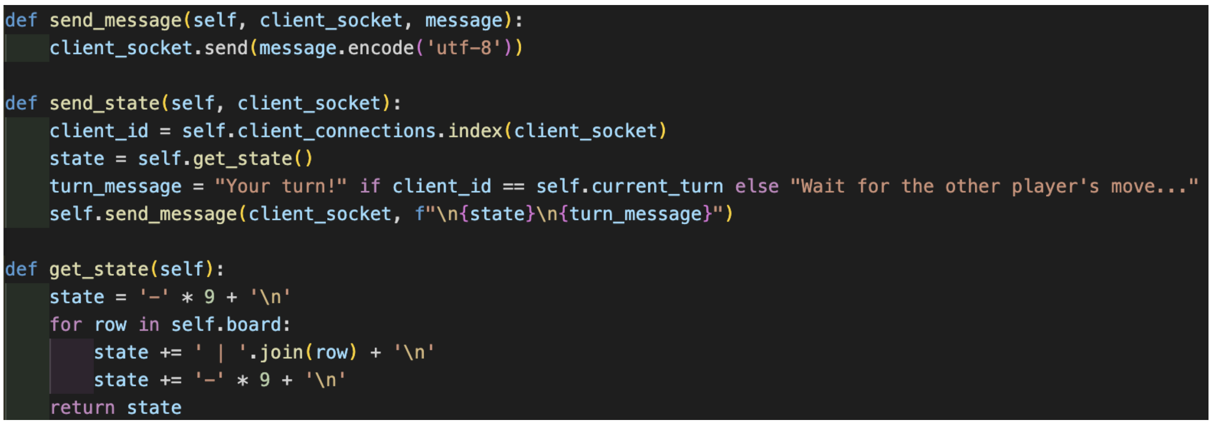
handle_client(self, client_socket): This method is responsible for handling a client's requests and managing the game flow for that client. It receives messages from the client and performs actions based on the message type. The messages from the clients can be a:

- ‘MOVE’: to process a move that can be an integer from 1 to 9.
- ‘RESTART’: to restart the game if both clients desire to do so.
- ‘QUIT’: to finish the game if either one of the clients wants to quit



```python
def process_move(self, client_socket, move):
    client_id = self.client_connections.index(client_socket)
    if client_id != self.current_turn:
        self.send_message(client_socket, "Invalid move. It's not your turn.")
        return
    if not move.isdigit() or not move:
        self.send_message(client_socket, "Invalid move. Move must be a number.")
        return
    try:
        move = int(move)
        row = (move - 1) // 3
        col = (move - 1) % 3
        print("Player {} placed a {} at position ({}, {})".format(client_id, self.player_symbols[client_id], row, col))
        if row < 0 or row > 2 or col < 0 or col > 2:
            raise ValueError

        if self.board[row][col] != ' ':
            raise ValueError

        self.board[row][col] = self.player_symbols[client_id]
        self.current_turn = 1 - self.current_turn

        if self.check_winner():
            print("Player", client_id, "wins!")
            self.score[client_id] += 1
            self.send_message(self.client_connections[client_id], "Congratulations! You won!")
            self.send_message(self.client_connections[1 - client_id], "You lost. Better luck next time!")
        elif self.check_tie():
            print("It's a tie!")
            self.send_message(self.client_connections[0], "It's a tie!")
            self.send_message(self.client_connections[1], "It's a tie!")
        else:
            self.send_state_to_all()
    except ValueError:
        print("Illegal move")
        self.send_message(client_socket, "Invalid move. Please change your move.")
```

process_move(self, client_socket, move): This method processes a move made by a client. First, it converts the input which is in the range [1,9] to x, y coordinates. It verifies if it is the client's turn and if the move is valid. If the move is valid, it updates the game board, switches the turn, checks for a winner or tie, and sends the updated state to all clients. Also, after each valid move, it checks if the game is finished or not.

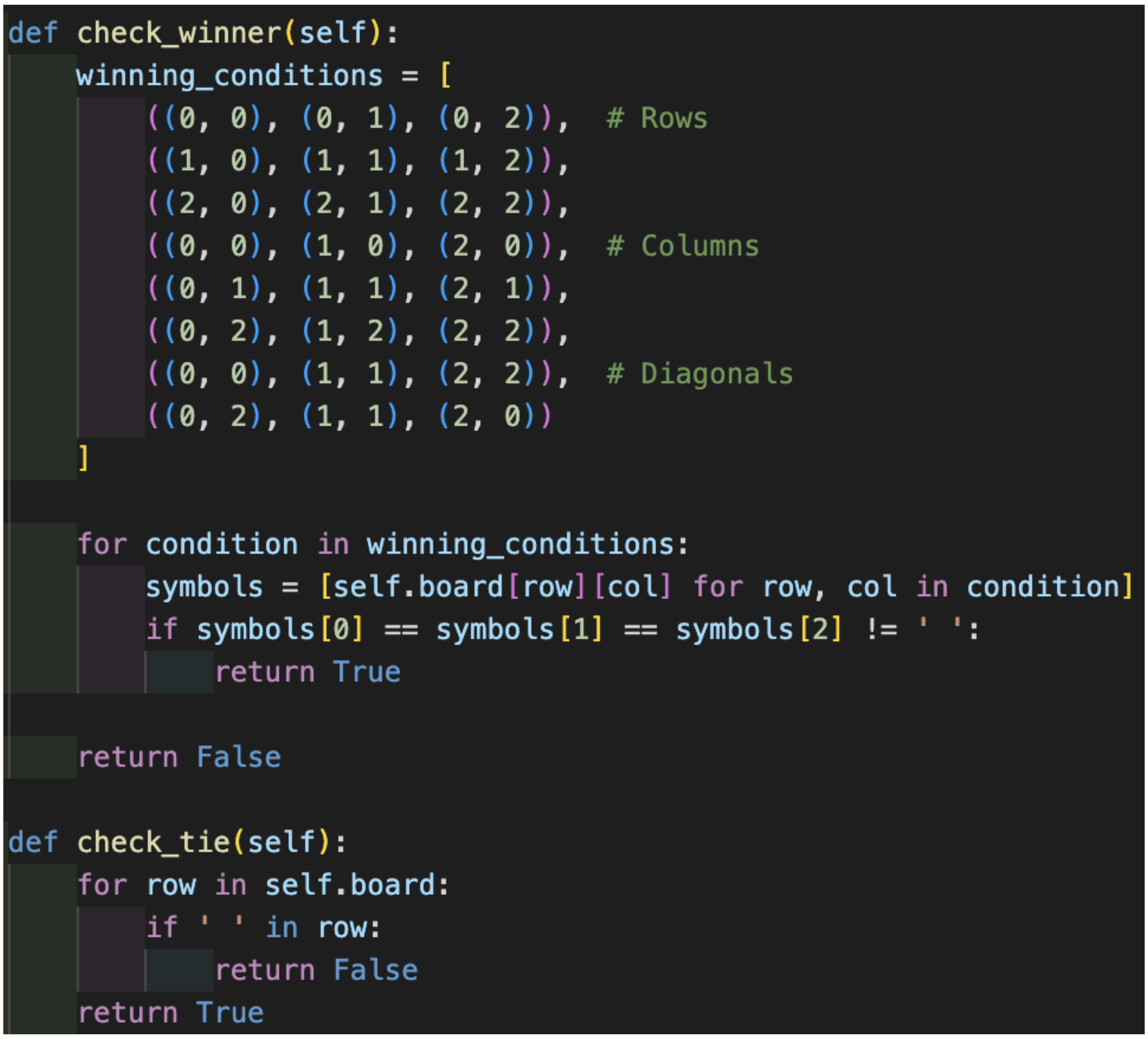


```python
def send_message(self, client_socket, message):
    client_socket.send(message.encode('utf-8'))

def send_state(self, client_socket):
    client_id = self.client_connections.index(client_socket)
    state = self.get_state()
    turn_message = "Your turn!" if client_id == self.current_turn else "Wait for the other player's move..."
    self.send_message(client_socket, f"\n{state}\n{turn_message}")

def get_state(self):
    state = '-' * 9 + '\n'
    for row in self.board:
        state += ' | '.join(row) + '\n'
        state += '-' * 9 + '\n'
    return state
```

send_message(self, client_socket, message): This method sends a message to a specific client using the client socket. The message is encoded as UTF-8 before sending.

send_state(self, client_socket): This method sends the current game state to a specific client. It retrieves the state using the get_state() method, determines the turn message based on the client's turn, and sends the state and message to the client.

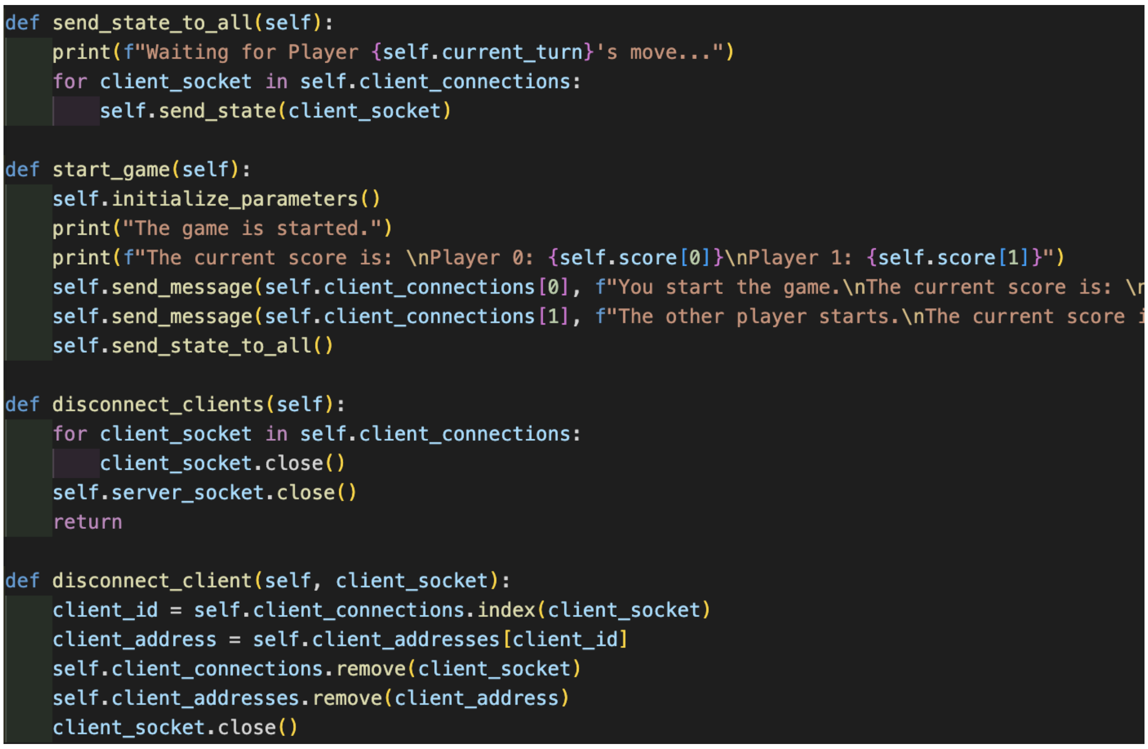
get_state(self): This method generates a string representation of the current game state. It iterates over the game board and constructs a formatted representation of the board using dashes and pipe characters.



```python
def check_winner(self):
    winning_conditions = [
        ((0, 0), (0, 1), (0, 2)),  # Rows
        ((1, 0), (1, 1), (1, 2)),
        ((2, 0), (2, 1), (2, 2)),
        ((0, 0), (1, 0), (2, 0)),  # Columns
        ((0, 1), (1, 1), (2, 1)),
        ((0, 2), (1, 2), (2, 2)),
        ((0, 0), (1, 1), (2, 2)),  # Diagonals
        ((0, 2), (1, 1), (2, 0))
    ]

    for condition in winning_conditions:
        symbols = [self.board[row][col] for row, col in condition]
        if symbols[0] == symbols[1] == symbols[2] != ' ':
            return True

    return False

def check_tie(self):
    for row in self.board:
        if ' ' in row:
            return False
    return True
```

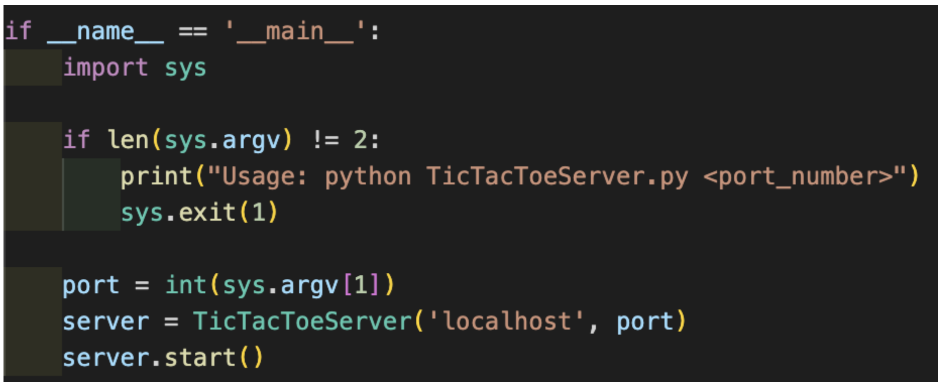
check_winner(self): This method checks if there is a winner in the game. It defines a list of winning conditions (rows, columns, and diagonals) and checks if any of these conditions have the same symbol ('X' or 'O').

check_tie(self): This method checks if the game has ended in a tie. It checks if there are any empty spaces left on the game board.



```python
def send_state_to_all(self):
    print(f"Waiting for Player {self.current_turn}'s move...")
    for client_socket in self.client_connections:
        self.send_state(client_socket)

def start_game(self):
    self.initialize_parameters()
    print("The game is started.")
    print(f"The current score is: \nPlayer 0: {self.score[0]}\nPlayer 1: {self.score[1]}")
    self.send_message(self.client_connections[0], f"You start the game.\nThe current score is: \r
    self.send_message(self.client_connections[1], f"The other player starts.\nThe current score i
    self.send_state_to_all()

def disconnect_clients(self):
    for client_socket in self.client_connections:
        client_socket.close()
    self.server_socket.close()
    return

def disconnect_client(self, client_socket):
    client_id = self.client_connections.index(client_socket)
    client_address = self.client_addresses[client_id]
    self.client_connections.remove(client_socket)
    self.client_addresses.remove(client_address)
    client_socket.close()
```

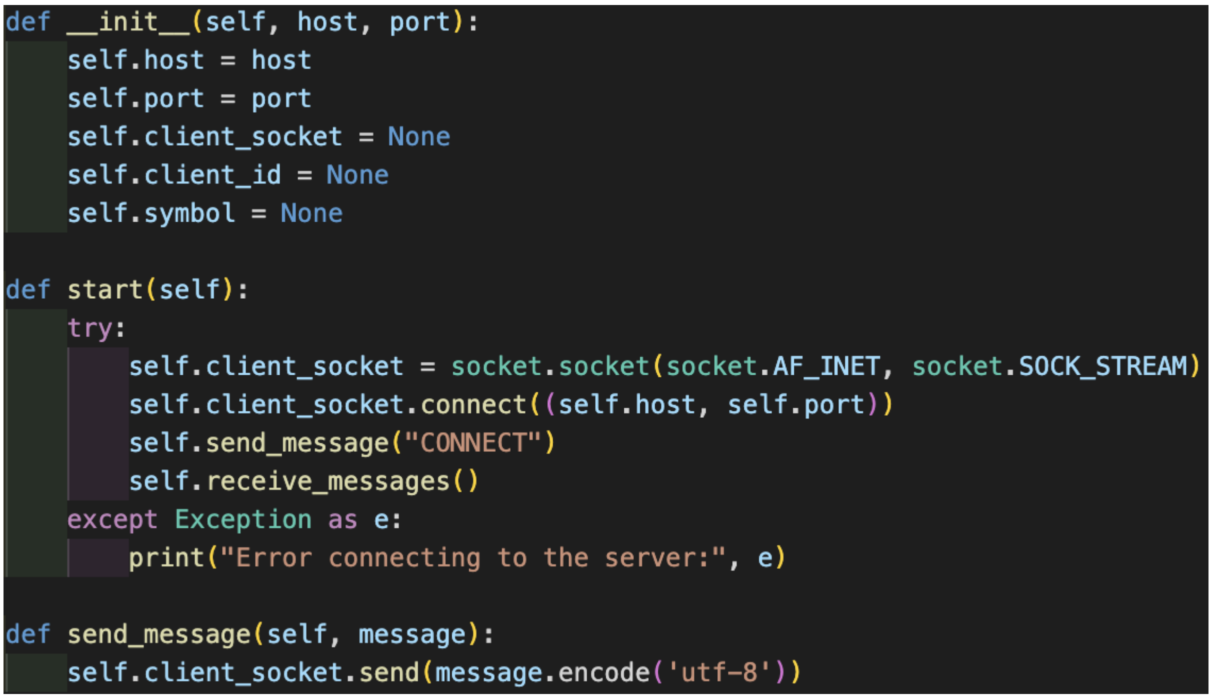
send_state_to_all(self): This method sends the current game state to all connected clients. It retrieves the state and turn message using the get_state() and send_state() methods, respectively, and sends them to each client.
start_game(self): This method starts a new game. It initializes the game parameters, sends the current score and starting message to the clients, and sends the initial game state to all clients.
disconnect_clients(self): This method disconnects all clients and closes their sockets. It iterates over the client connections, closes each socket, and removes the client from the lists. Finally, it closes the server socket.
disconnect_client(self, client_socket): This method disconnects a specific client and removes it from the client connections and addresses lists. It closes the client socket.



```python
if __name__ == '__main__':
    import sys

    if len(sys.argv) != 2:
        print("Usage: python TicTacToeServer.py <port_number>")
        sys.exit(1)

    port = int(sys.argv[1])
    server = TicTacToeServer('localhost', port)
    server.start()
```

main (driver): It retrieves the port number from the command-line arguments, creates an instance of the TicTacToeServer class, and calls the start() method to begin the server.

Client Side Implementation

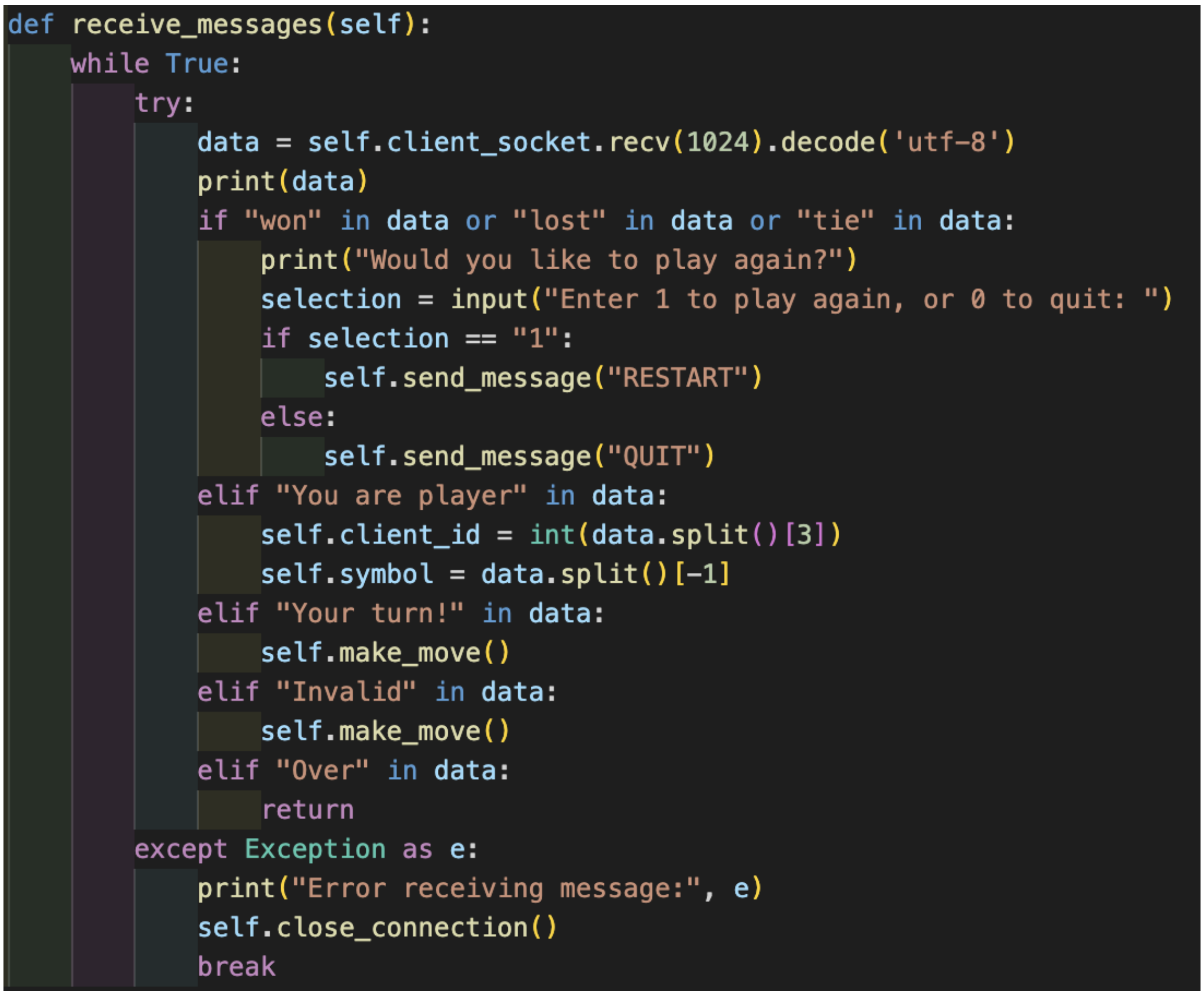


```python
def __init__(self, host, port):
    self.host = host
    self.port = port
    self.client_socket = None
    self.client_id = None
    self.symbol = None

def start(self):
    try:
        self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.client_socket.connect((self.host, self.port))
        self.send_message("CONNECT")
        self.receive_messages()
    except Exception as e:
        print("Error connecting to the server:", e)

def send_message(self, message):
    self.client_socket.send(message.encode('utf-8'))
```

__init__(self, host, port): This is the constructor method for the client class. It initializes various attributes, such as the host and port for the server, the client socket, client ID, and symbol.

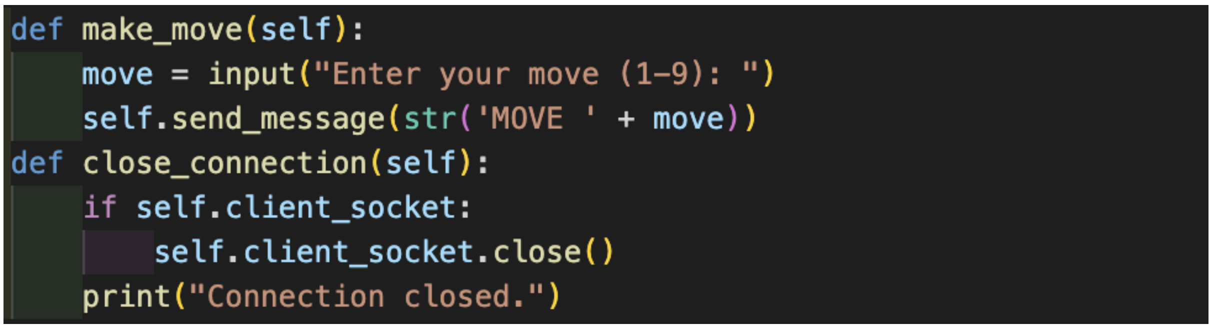
start(self): This method is the entry point for starting the client. It creates a client socket, connects it to the specified host and port, sends a "CONNECT" message to the server, and begins receiving messages from the server.

send_message(self, message): This method sends a message to the server using the client socket. The message is encoded as UTF-8 before sending.



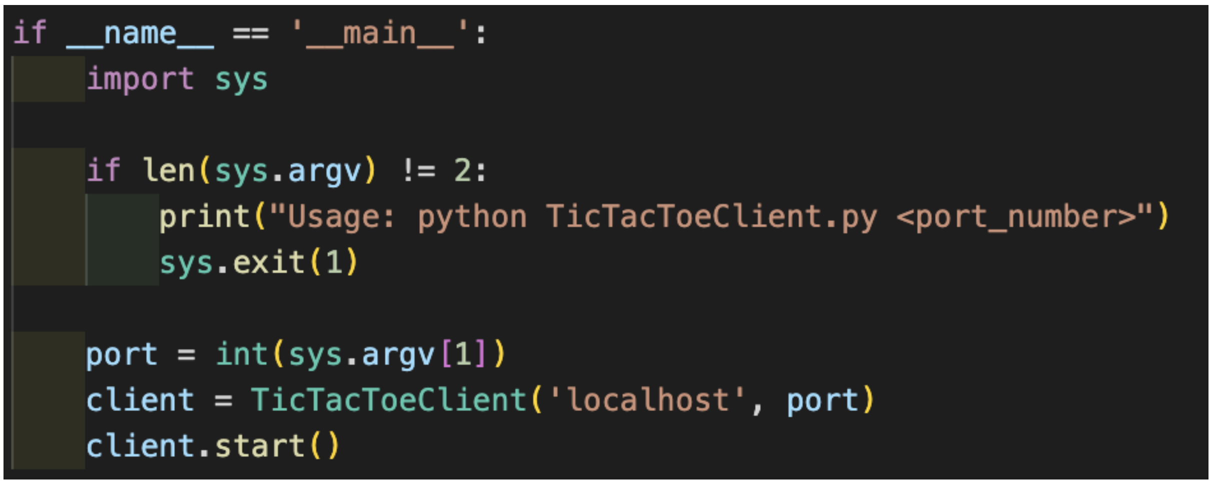
```python
def receive_messages(self):
    while True:
        try:
            data = self.client_socket.recv(1024).decode('utf-8')
            print(data)
            if "won" in data or "lost" in data or "tie" in data:
                print("Would you like to play again?")
                selection = input("Enter 1 to play again, or 0 to quit: ")
                if selection == "1":
                    self.send_message("RESTART")
                else:
                    self.send_message("QUIT")
            elif "You are player" in data:
                self.client_id = int(data.split()[3])
                self.symbol = data.split()[-1]
            elif "Your turn!" in data:
                self.make_move()
            elif "Invalid" in data:
                self.make_move()
            elif "Over" in data:
                return
        except Exception as e:
            print("Error receiving message:", e)
            self.close_connection()
            break
```

receive_messages(self): This method continuously receives messages from the server and handles them accordingly. It decodes the received data, prints it, and checks for specific messages such as winning/losing/tie, player assignment, turn notification, invalid move, or game over. It prompts the user to play again or quit based on the game outcome.



```python
def make_move(self):
    move = input("Enter your move (1-9): ")
    self.send_message(str('MOVE ' + move))
def close_connection(self):
    if self.client_socket:
        self.client_socket.close()
    print("Connection closed.")
```

make_move(self): This method prompts the client to enter their move (a number between 1 and 9) and sends it to the server as a "MOVE" message.
close_connection(self): This method closes the client socket and prints a message to indicate that the connection has been closed.


```python
if __name__ == '__main__':
    import sys

    if len(sys.argv) != 2:
        print("Usage: python TicTacToeClient.py <port_number>")
        sys.exit(1)

    port = int(sys.argv[1])
    client = TicTacToeClient('localhost', port)
    client.start()
```
main (driver): It retrieves the port number from the command-line arguments, creates an instance of the TicTacToeClient class, and calls the start() method to begin the client.