Nan Chen

The summary of the files:

In the server.py, it manages a two-player game using sockets for network communication. It handles multiple connections by using threads and manages game states with the Game class.

In the client.py, it connects to a server, handle user input, and update the game state based on the server responses. The game allows two players to play Tic Tac Toe remotely, with each player interacting with the game through their own client application.

In the network.py, it contains the “Network” class. It is used by the client-side Tic Tac Toe game to handle the network communication with the server. The client can create an instance of the Network class, connect to the server, send game data, and receive updates from the server using the methods defined in this class.

In the game.py: it contains the “Game” class, which is responsible for handling the game logic, such as checking for winners, validating moves, recording the moves, resetting the game, and managing player choices.

In the button.py: it defines the “Button” class, which is a simple button implementation for Pygame. It has methods for drawing the button on the screen, and detecting if the button is clicked based on the user's mouse position.

In the board.py: it defines the “Board” class, which is used to create an instance of the board for a game, handle drawing and updating the board, and process user clicks to determine the corresponding cell.

In the main.py: it serves as an entry point to start either the server or the client side of a Tic Tac Toe game. It takes command-line arguments to determine whether to start the server or the client and to specify the necessary parameters (hostname and port) for the connection.

The details of actions in the Game:

1. A common board configuration will be used, and both sides must begin with a clear board.

   To make a common board configuration and ensure both sides begin with a clear board, I create a shared game state that both players will have access to. The shared game state that contains the board's state (e.g., an empty list containing nine empty spaces). In the client.py, I import the Board class and create an instance of the Board class and initialize the Board class with the desired position and cell size. The “redrawWindow” function is responsible for updating the game window. It checks if the game is connected or if the first player has been determined, and then it renders the necessary game elements accordingly. In the “redrawWindow” function, it calls board.draw() with the current game state. When drawing the

board, the board.draw() method is called, which takes win (the window surface) and game.get_board() as parameters. The game.get_board() method should return the current game state, which should initially be an empty board (e.g., a list containing nine empty spaces, like [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']). Therefore, both clients start with the same empty board configuration and can update the board as they play.

2. The two sides must negotiate which side will be X player and which side will be O player.

   When a new game is initiated, game.get_first_player() returns -1 since the choices list is not yet determined (initially set to [None, None]). This means the negotiation process has not yet started. The negotiation window is displayed to both clients, presenting them with "agree" and "disagree" buttons. The message on the window indicates the first client to join (player 0) will be 'X' player if they both agree, and the second client (player 1) will be 'O' player. Each client can either click "agree" or "disagree" to the proposed sides (X or O). When a client clicks a button, their choice is sent to the server using n.send("agree") or n.send("disagree"). The server then updates the choices list in the game state accordingly. For example, if the first client clicks "agree" and the second client clicks "disagree", the choices list will be ["agree", "disagree"]. The code checks if the choices list has one "agree" and one "disagree" element. If it does, the server resets the choices list back to [None, None] by calling n.send("resetChoices"). This allows the clients to negotiate again. If both clients click "agree", the choices list will be ["agree", "agree"], and game.get_first_player() will return 0, meaning the first client (player 0) will be 'X' player, and the second client (player 1) will be 'O' player. If both clients click "disagree", the choices list will be ["disagree", "disagree"], and game.get_first_player() will return 1, meaning the first client (player 0) will be 'O' player, and the second client (player 1) will be 'X' player. Once the negotiation is complete and game.get_first_player() is no longer -1, the actual game starts, and the clients will see the game board along with their assigned sides (X or O).

3. State of which player has a turn needs to be maintained.

   In the Game class (in game.py), there is an attribute current_player which is initialized to "X". This attribute will keep track of which player's turn it is, either "X" or "O". When a client successfully makes a valid move, the play method in the Game class is called. This method updates the game board with the player's move, adds the move to either p1Moves or p2Moves (depending on the player), and then updates the current_player attribute to the next player's symbol ("X" or "O"). In the client.py file, inside the main function, the game object is received from the server. This object contains the updated current_player attribute. When processing the MOUSEBUTTONDOWN event in the client, the code checks if the clicked position is a valid move with game.check_valid_move(clicked_position), and if the current player matches the local player with game.current_player == p. If both conditions are met, the client sends the move to the server by calling n.send(str(clicked_position)). The server processes the move, updates the game state, and sends it back to both clients. The clients update their local game objects with the new state, which includes the updated current_player attribute, indicating whose turn it is. By following this process, the state of which player has a turn is maintained

and updated in the current_player attribute of the Game class, which is then shared between the server and both clients.

4. Communicating the move needs to be defined.

   Both clients connect to the server, and the server assigns a player number (0 or 1) to each client. The clients run a loop where they listen for events, such as button clicks, and send relevant data to the server based on the actions performed. When a player makes a move, the client sends the move's position to the server. For example, if a player clicks on a position on the game board, the client sends that position to the server. The server-side script receives the data and processes it. In the "threaded_client" function, it checks if the received data corresponds to a move. If so, the server calls the "play()" method of the Game class, which updates the game state accordingly. After processing the data, the server sends the updated game state back to both clients using "pickle.dumps(game)". This serializes the game object and sends it over the network. Clients receive the updated game state and update their local game state accordingly. They redraw the game window to reflect the new state. The process repeats for every move until the game ends or one of the clients disconnects.

5. Rewriting the gameboard and ensuring that they are aligned, though independent, requires communication and alignment.

   On the client side, the main function is responsible for handling the game loop. During each iteration of the game loop, the client fetches the current game state by sending a "get" request to the server (game = n.send("get")). The server-side threaded_client function receives the "get" request, retrieves the game instance associated with the client, and sends the game state back to the client using conn.sendall(pickle.dumps(game)). The game state is serialized using the pickle module. The client receives and deserializes the game state (game = n.send("get")). This ensures that both clients have the most recent game state. The client's redrawWindow function uses the deserialized game state to update and redraw the game board. This function takes care of handling the different game states, such as waiting for a player, negotiating the first player, and updating the board after each move. When a client makes a move, the main function sends the move to the server (n.send(str(clicked_position))). The server updates the game state using the play method of the Game class (game.play(p, data)). The updated game state is then sent back to both clients during their next "get" request. This ensures that both clients have an aligned view of the game board.

6. Declaring a winner needs to be identified and communicated.

   The game logic for determining the winner resides in the Game class in game.py. In the winner method, it checks whether the set of moves made by player 1 or player 2 forms a winning combination (win_positions). If a winning combination is found, it returns the index of the winning player (0 or 1). On the server side, in the threaded_client function, the server processes the received data and, depending on the command, performs the appropriate action on the game object. If the client sends a move (not "get" or any other command), the server updates

the game state by calling the play method. After updating the game state, the server sends the updated game object back to the clients using pickle.dumps(game). On the client side, in the main function, the clients receive the updated game object and redraw the window using the redrawWindow function. Inside the redrawWindow function, the game object's game_over method is called to check if the game is over. If the game is over, the replayBtn is drawn, and the result is displayed using the draw_message function. The winner is determined by calling the winner method on the game object, which returns the index of the winning player (0 or 1). The clients then compare the returned winner index with their player index to decide if they won, lost, or if it's a tie.

7. Replaying the game and resetting the game environment should take place.

   If the game is over, the replayBtn is drawn. In the client.py file, when the replayBtn is clicked, the client sends a "reset" message to the server. In the server.side file, the server receives the "reset" message and calls the reset_game() method on the Game object. In the game.py file, the reset_game() method of the Game class resets the game environment by resetting the game board, clearing the sets that store the moves of each player, setting the current player back to "X", resetting the choices of both players, setting the exit flag to False. After these steps are executed, the game environment is reset, allowing the players to start a new game while maintaining the same connection.

8. Ending and closing the program gracefully needs to be designed.

   In the client.py main function, the code listens for a pygame.MOUSEBUTTONDOWN event. If either the choiceExitBtn or exitBtn button is clicked, the code sends the 'exit' message to the server by calling n.send('exit'). On the server side, in the threaded_client function, it checks for incoming data from the client. If the data is equal to 'exit', the server calls the game.exit_game() method to set the exit attribute of the Game object to True. In the client.py main function, the code checks if the game.exit attribute is set to True. If it is, the game window is filled with a solid color, a "Someone Exit. Game Stop..." message is displayed, and the pygame.display.update() method is called to update the game window.
   After displaying the message, the code waits for 4 seconds using pygame.time.delay(4000), then sets the run variable to False, breaking out of the main while loop. The pygame.quit() method is called to clean up and close the Pygame window. On the server side, when the client disconnects, it reaches the print("Lost connection") line, decrements the idCount variable, deletes the game from the games dictionary, and closes the connection using conn.close().