CMPT 371: Final Project group 17

Date: August 14th 2022

Group Members:
Khanh Bui
Clare Dang
Anna Liang
Kyle Locquiao
Damir Paripovic

Links

Link to working demo of project: https://youtu.be/-hq5VQF1p5E

Link to github repo: https://github.com/klocquiao/Trivia-Game/tree/main

Description

The game we built is a trivia game. Four players are required to play and the game will begin after all four players are connected to the server. The game consists of 5 rounds, each of which consists of 3 turns. At the start of each round, the players are prompted with a question and a 4 x 4 grid of answers. The answers presented to the player include 9 correct answers and 7 incorrect answers. Each player can choose one answer per turn (if the answer is correct the player gets 1 point). Once a player chooses an answer, that answer becomes grayed out and no other players can click on that answer for the remainder of the round. A turn is complete once each player has chosen an answer. Once 3 turns are completed, the round is completed. Once all 5 rounds are completed the winner is announced at the end game screen.The game requires the server to be started on a computer and then all players can connect to the server.

The game was written in python and requires the Pygame library to present the players with the frontend display. The game code was split into client (frontend) code and server (backend) code.

The client code handles what each player sees and interacts with. The players are able to submit the name they wish to be identified by and when the game starts, the players can interact with the answers by clicking on them. The code was broken down into multiple parts that handle:

- the lobby and initializing the game space (screen and required structures)
- layout that transforms information from server to what a player sees
- the client side logic and networking needs (message exchange)
- the player object and all of its attributes and methods

For example, once a client establishes connection and 3-way handshake successfully, the server continues to look for players(find_players()) and creates a player object and passes it to Player_manager to control. Server starts to receive clients' requests and also to send data to clients.

In the Lobby page(game opening), after a player finishes name entering and clicks the "Ready" button, get_player_name() passes the object player's name to client handle_message().Once handle_message() receives the token of "name" from the server and it sends it back to the server. Also, a client establishes connection to the server once "Ready" button is clicked.

In the waiting page, players will stay at the waiting page until the server receives 4 player connections. It sends out the token of “round” to all clients and starts the game.

Game over page - once all rounds are done, the game is finished. Server calculates the highest points and sends out a token of “winner” to all clients. Lastly, game page swaps to game over page and it displays the winner's name get_winner_name().

The figure below (Fig 1) shows the creation of a client side socket. Also shown is how the client side sends the application level messages and how the received messages from the server are handled. The receiver_runner() function handles the received messages.

```
def receiver_runner():
    while True:
        try:
            data = my_socket.recv(MAX_MESSAGE_SIZE).decode("utf-8")
            handle_message(data)
        except:
            print("An error has occured!")
            my_socket.close()      ■ "close" is not a known member of
            break

# get player's name from lobby
def new_player(pname):
    global player_name
    player_name = pname

def handle_message(data):
    message = json.loads(data)
    if message["token"] == "Name":
        # tm = {"token": "Name", "name": "ClareKyleDamirAnnaKhanh"} #
        tm = {"token": "Name", "name": player_name}
        send_message(tm)

# To be binded by front-end team members
def send_message(message):
    data = json.dumps(message)
    my_socket.sendall(bytes(data,encoding="utf-8"))      ■ "sendall" i

# Be called in lobby
def start_client():
    global my_socket
    my_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    my_socket.connect((HOST, PORT))

    print("Connected to server!")
    receiver_thread = Thread(target=receiver_runner)
    receiver_thread.start()
```

Fig 1: Client code. Shows the creation of the socket and how messages are handled.

The server code is broken up into objects, and the corresponding object managers, as well as the server logic code (which is broken into a game file and a server file). The different objects are answer, player, round, trivia, and the managers are the trivia_manager, player_manager. The game object initializes all of the required objects and starts the server. The server file contains the code that will handle requests from players and update players on any changes to the game states. The below figure (Fig 2) shows the answer class and how it handles the shared objects. Each shared object is an instance of the answer class and can access the mutex to Lock access to the board temporarily. The player that clicks on a specific answer first gets to gray out that answer and changes the "is_available" flag from True to False. So no other players may access or use that answer.

```
from threading import Thread, Lock

mutex = Lock()

class Answer:
    def __init__ (self, answer, is_correct):
        self.answer = answer
        self.is_correct = is_correct
        self.is_available = True

    def __str__(self):
        return self.answer

    def check_available(self):
        mutex.acquire()
        temp = self.is_available
        if self.is_available:
            self.is_available = False
        mutex.release()

        return temp

    def check_is_correct(self):
        return self.is_correct
```

Fig 2: Server code in the answer object that handles concurrency when multiple players attempt to access the same object.

Below in Fig 3, we can see the creation of the server side sockets and along with Fig 4 the scheme for handling requests from the players that are connected to the server. After receiving a request from a player in the "receiver_runner()" function, the server will attempt to handle the request from the player in the "handle_message()" function and call the appropriate methods on the game object. This design also allows for further expansion if

more complexity is required since the varying requests to the server can be placed inside of the handle_message() function. The server can also broadcast a message to all of the players or send a message to a specific player. This can be seen below in Fig 3.

```
def handle_message(data):
    message = json.loads(data)
    if message["token"] == "Answer":
        game.current_round.check_player_answer(message)      ■ "cur

def broadcast_message(message):
    data = json.dumps(message)
    for player in game.player_manager.get_players():      ■ "player
        player.get_socket().sendall(bytes(data,encoding="utf-8"))


def send_message(player, message):
    data = json.dumps(message)
    player.get_socket().sendall(bytes(data,encoding="utf-8"))

def start_server(new_game):
    global game
    global my_socket

    game = new_game
    my_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    my_socket.bind((HOST, PORT))
    my_socket.listen()

    find_players()

def close_server():
    my_socket.close()      ■ "close" is not a known member of "None
```

Fig 3: Functions from the server file code showing the creation of the server side sockets.

```
def receiver_runner(client):
    while True:
        try:
            data = client.recv(MAX_MESSAGE_SIZE).decode("utf-8")
            handle_message(data)
        except:
            print("A client has disconnected! Exiting game...")
            my_socket.close()      ■ "close" is not a known member
            break
```

Fig 4: The receiver_runner() function from the server file code, depicting how the server handles requests from clients.

Messaging Scheme:

Token: Name (server -> client)
Parameters: N/A
- Server sends this after the client establishes a connection to request a name from the player.

Token: Name (client -> server)
Parameters: name
- Client sends this after the server requests it to provide a name after establishing a connection.

Token: Players (server -> client)
Parameters: players (array of player names)
- Server broadcasts this after all players have connected so that the client side has a copy of the players in the game.

Token: Round (server -> client)
Parameters: number (current round #), question, answers
- Server broadcasts this after the start of each round to inform the client to start a new ui with a new trivia set.

Token: Turn (server -> client)
Parameters: number
- Server broadcasts this at the start of each round. Should unlock each player to allow them to choose again.

Token: Player (server -> client)
Parameters: name, score, answer
- Server broadcasts this after the client successfully chooses an answer. All clients should update their players to reflect this change. This message should also lock the answer that the specific player has chosen for each other client.

Token: Locked (server -> client)
Parameter: N/A
- Server sends to the client who tried to choose a locked answer to unlock and tell them to pick another answer.

Token: Winner (server -> client)
Parameter: name
- Server broadcasts this after the game is complete. It contains the calculated winner for the client to display.

Token: Answer (Client -> Server)
Parameter: answer, name
- Client sends this whenever they click an answer button on their screen. The server verifies this answer.

Member contribution

Khanh Bui - 22
Clare Dang - 22
Anna Liang - 22
Kyle Locquiao - 22
Damir Paripovic - 12