ONLINE CAR RACING GAME

Ain Shams University – Faculty of Engineering

Computer Engineering and Software Systems

CSE354: Distributed Computing Project Report

Table of Contents

Team Members:

		Roles
Adham Ehab Salman Selim	19p4388	Chat, Hosting, Documentation
Yassin Khaled Mostafa Attia Mahgub	19p9597	Video Game
Gannah Allah Mohamed Gaber	19p9610	Video Game
Ziad Ashraf Ahmed Ahmed	19p7095	Video Game

Introduction

Within this group project, the team was successfully able to build an online car racing game and a chat room for the connected clients. Moreover, a backup server is implemented in case of any failure within the main server all the connected clients are migrated to the backup server.

The project is developed using Python libraries such as PyGame and Tkinter. Regarding hosting the servers online, AWS EC2 was used to create two instances which contains the main server and the backup server, so the clients can connect to the servers using the internet.

Project Description

The project can be broken down into several components:

- Client’s chat
- Chat server
- Client's video game
- Video game server
- Hosting on AWS
- Backup and Migration

Client's chat:

First of all, when running the client file the client is connected to the server and a simple dialogue is shown in which the client should enter a unique user name the will identify him through the following process, afterwards two threads are started one responsible for generating the GUI and sending the chat messages, and the other thread is responsible for receiving any incoming messages.

```
class Client:

    def __init__(self,host, port):

        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect((host,port))
        msg = tkinter.Tk()
        msg.withdraw()
        self.nickname = simpledialog.askstring("Nickname", "Please Choose a nickname", parent=msg)
        self.gui_done = False
        self.running = True

        gui_thread = threading.Thread(target=self.gui_loop)
        receive_thread= threading.Thread(target=self.receive)

        gui_thread.start()
        receive_thread.start()
```

Figure 1

Whenever the send button is clicked the write function is called which is responsible for building, encoding, and sending the message.

```
def write(self):
    message = f"{self.nickname}: {self.input_area.get('1.0', 'end')}"
    self.sock.send(message.encode('utf-8'))
    self.input_area.delete('1.0', 'end')
```

Figure 2

Regarding the receiving thread, first a while true loop is created to loop on the sock.recv and the and the handling of the received message.

```
def receive(self):
    print("started receiving")
    while self.running:
        try:
            message = self.sock.recv(1024).decode('utf-8')
            special_message = message.split('$')
            if message == "":
                raise Exception

            for m in special_message:
                splitted_msg = m.split(' ')
                print(m)

                if splitted_msg[0] == "NEWCONN":
                    connected_clients = []
                    for x in range(1, len(splitted_msg)):
                        connected_clients.append(splitted_msg[x])
                    print(connected_clients)
                elif m == "NICK":
                    self.sock.send(self.nickname.encode('utf-8'))
                elif splitted_msg[0] == "CHAT":
                    if self.gui_done:
                        splitted_msg.remove("CHAT")
                        m = ""
                        for each in splitted_msg:
                            m = m + " " + each

                        self.text_area.config(state='normal')
                        self.text_area.insert('end', m)
                        self.text_area.yview('end')
                        self.text_area.config(state='disabled')

        except ConnectionAbortedError:
            break
        except:
            print("Main server crashed")
            self.sock.close()
            self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.sock.connect((self.backup_port, self.backup_port))

            print("connected to backup server")
```

Figure 3

Each message sent by the server have a dollar sign in the end to be able to separate the messages from each other because if more than one message is sent from the server at once they may be received as one, so they are split using the dollar sign and handled separately. There are three types of messages that can be received by the client regarding the chat feature. First, the NEWCONN message which identifies a message that represents a connection of a new client to the server, this message contains a list of the clients connected to server and is stored within another list at the client's file when this message is received. Second, the NICK message, this message is sent by the server to request the nickname of the client and the nickname will be sent when this message is received. Third, the CHAT messages, these messages are decoded and added to the chat box to be displayed after removing the head of the message.

There are two exceptions that can be raised in the receive thread. First, an exception is raised manually when an empty message is received because when the server crashes sometimes the host machine keeps spamming empty messages. Second, a connection aborted error can be thrown automatically if the connection with the server is interrupted. In both cases, the client will close this socket and establish a new connection with the backup server.

Chat server:

The main purpose of the chat server is to announce a new connection to all the clients previously connected and to broadcast the messages received to all the connected clients.

```
def receive():
    while True:
        client, address = server.accept()
        print(f"Connected with {str(address)}!")
        client.send("NICK".encode('utf-8'))
        nickname = client.recv(1024).decode('utf-8')
        print(f"client's name is " + nickname)
        connected_clients="NEWCONN"


        clients.append(client)
        nicknames.append(nickname)
        for i in nicknames:
            connected_clients = connected_clients + " "+ i
        print(connected_clients)
        connected_clients = connected_clients+"$"
        broadcast(connected_clients.encode('utf-8'))

        broadcast(f"CHAT {nickname} connected to the server!\n$".encode('utf-8'))
        client.send("CHAT You are now connected to the server\n$".encode('utf-8'))
        thread = threading.Thread(target=handle, args=(client,))
        thread.start()
```

Figure 4

The receive function is responsible for accepting the new connections and announcing it then the new client is dispatched to a new thread responsible for handling this client only, so each new client is being handled by a thread in a one-to-one relationship.

```python
def broadcast(msg):
    for client in clients:
        client.send(msg)

def handle(client):
    while True:
        try:
            message = client.recv(1024)
            print(f"{nicknames[clients.index(client)]}: {message}")
            broadcast(message)
        except:
            index = clients.index(client)
            clients.remove(client)
            client.close()
            nickname = nicknames[index]
            nicknames.remove(nickname)
            break
```

Figure 5

The handle function formats the message to be suitable for printing and broadcasts is to all the client if any exception is thrown the connection is closed with that client to prevent crashing down of the whole server, so disconnecting a single client would be better that disconnecting all those who are connected.

```python
def update_backup():
    try:
        sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        sock.connect(("13.51.48.183",5561))
        while True:
            sock.send("backup".encode('utf-8'))
            time.sleep(2)
    except:
        print("backup server not running")
```

Figure 6

This function runs on a separate thread which periodically updates the backup server with its status, if the server failed to connect to the backup server it is assumed that it is not running.

Client's Video Game:

The video game is developed using python's PyGame library to make a simple 2D car dodging racing game and the first one to reach 1000 points wins the race.

```
def initialize(self,nick):
    self.crashed = False

    self.car1Img = pygame.image.load('.\\img\\car.png')
    self.car_width = 49

    # enemy_car
    self.enemy_car = pygame.image.load('.\\img\\enemy_car_1.png')
    self.enemy_car_startx = random.randrange(310, 450)
    self.enemy_car_starty = -600
    self.enemy_car_speed = 5
    self.enemy_car_width = 49
    self.enemy_car_height = 100

    # Background
    self.bgImg = pygame.image.load(".\\img\\back_ground.jpg")
    self.bg_x1 = (self.display_width / 2) - (360 / 2)
    self.bg_x2 = (self.display_width / 2) - (360 / 2)
    self.bg_y1 = 0
    self.bg_y2 = -600
    self.bg_speed = 3
    self.count = 0

    self.server_host = "16.16.27.193"  # Replace with the server's host address
    self.server_port = 5560  # Replace with the server's port number
    self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.server.connect((self.server_host, self.server_port))
    self.nickname = nick
    self.server.send((self.nickname).encode())
    self.initialvalues = self.server.recv(1024).decode()

    self.initialvalues = self.initialvalues.split("|")
    print(self.initialvalues)
    self.car_x_coordinate, self.car_y_coordinate = self.read_pos(self.initialvalues[2])
    self.count = int(self.initialvalues[1])
    print("======================================")
```

Figure 7

The initialize function is used to initialize the main attributes such as the vehicles coordinates and the background racetrack. Afterwards, it connects to the game's server and sends the player's name to the server to be saved in a dictionary along with his car's coordinates.

```
def racing_window(self):
    self.gameDisplay = pygame.display.set_mode((self.display_width, self.display_height))
    pygame.display.set_caption('Car Dodge')

    # Start a thread to receive updates from the server
    receive_thread = Thread(target=self.receive_updates)
    otherClients_thread = Thread(target=self.renderOtherClients)
    receive_thread.start()
    otherClients_thread.start()
    self.run_car()
```

Figure 8

The racing window function is responsible for starting the game mainly along with two threads that are responsible for receiving the other cars coordinates to update and store them within the available cars dictionary and the other thread is responsible for reading from that dictionary to update the cars positions in game. The run car function is responsible for handling the main player car only and updating its coordinates within the server.

```
def renderOtherClients(self):
    print("hello from the thread")
    temp_list = []
    while True:
        left = 0
        right = 30
        if len(temp_list) == 0 and len(self.availableCars.keys()) != 0:
            newClient = Client_car()
            print("dict as list ")
            newClient.tag = list(self.availableCars.keys())[0]
            newClient.client_car = pygame.image.load('.\\img\\car2.png')
            temp_list.append(newClient)
            thingx, thingy = self.read_pos(self.availableCars[newClient.tag])
            self.gameDisplay.blit(newClient.client_car, (thingx, thingy))



        else:
            if len(self.availableCars) > len(temp_list):
                diff = len(self.availableCars) - len(temp_list)
                keyList = list(self.availableCars.keys())
                for i in range(len(keyList) - diff, len(keyList)):
                    newClient = Client_car()
                    newClient.tag = keyList[i]
                    newClient.client_car = pygame.image.load('.\\img\\car2.png')
                    temp_list.append(newClient)
            font = pygame.font.SysFont("arial", 20)
            for k in temp_list:
                thingx, thingy = self.read_pos(self.availableCars[k.tag])
                self.gameDisplay.blit(k.client_car, (thingx, thingy))
                text = font.render(k.tag + " Score : " + str(self.availableCarsScore[k.tag]), True, self.white)
                self.gameDisplay.blit(text, (left, right))

                right += 20
```

Figure 9

```
def receive_updates(self):
    # while not self.crashed:
    while True:
        try:
            # Receive updates from the server

            data = self.server.recv(1024).decode()
            if data == "":
                raise Exception("empty message received")

            splitted_data = data.split('$')
            for each_data in splitted_data:
                if each_data == '':
                    splitted_data.remove(each_data)
            print(splitted_data)

            for each_data in splitted_data:
                print("data received from server ", each_data)
                if len(each_data.split('|')) == 3:
                    each_data = each_data.split('|')
                    carID = each_data[0]
                    score = each_data[1]
                    if int(score) >= 1000:
                        self.display_message(f"{carID} won, game end")
                        self.connection = False
                        sleep(3)
                        pygame.quit()
                        exit()

                    pos = each_data[2]
                    self.availableCarsScore[carID] = score

                    self.availableCars[carID] = pos
                else:
                    print(each_data, " bazet hena ")
                    pos = each_data

                print("printing the dictionary", self.availableCars)

        except Exception as e:
            # Handle server disconnection

            print(e)
            print("Disconnected from the server")
            self.connection = False
            pygame.quit()
            exit()
            break
```

Figure 10

Finally when the client's file is being run to start both the chat and the game together each is dispatched to separate thread so they can work in parallel without interrupting each other. Each of them is communicating with a separate server.

```
if __name__ == '__main__':
    msg = tkinter.Tk()
    msg.withdraw()
    nickname = simpledialog.askstring("Nickname", "Please Choose a nickname", parent=msg)
    car_racing = CarRacing(nickname)
    game_thread = Thread(target=car_racing.racing_window)
    chat_client = ChatClient(nickname)
    chat_thread = Thread(target=chat_client.start_chat)
    game_thread.start()
    chat_thread.start()
```

Figure 11

First the user is asked for a nickname to be used for the chat and the racing game and then this name is passed to the constructors of the new objects the chat client and the car racing and both are launched on separate threads.

Video Game Server:

```
def start(self):
    print("Server started. Waiting for connections...")
    while True:
        client, address = self.server.accept()
        print("Connected to:", address)
        print("Connected to:", client)
        nickname= client.recv(1024).decode()
        print(f"{nickname} ba2a connected")
        #client.sendall("Welcome to the game!".encode())
        if nickname not in self.serveravailable:
            # Create a new car for the client
            car = Car()
            self.cars.append(car)
            self.myScore = 0
        else:
            data= self.serveravailable[nickname]
            self.myScore = self.serverscore[nickname]
            coords = data
            car_x, car_y = self.read_pos(coords)
            car = Car()
            self.cars.append(car)
            car.car_x_coordinate=car_x
            car.car_y_coordinate=car_y

        # Send the initial car coordinates to the client
        coordStr = self.make_pos(car.car_x_coordinate, car.car_y_coordinate)
        toBeSent = str(nickname) + "|" + str(self.myScore) + "|"+ coordStr
        #client.sendall(self.make_pos(car.car_x_coordinate, car.car_y_coordinate).encode())
        client.sendall(toBeSent.encode())
        # Start a new thread to handle the client
        thread = threading.Thread(target=self.handle_client, args=(client, car,nickname))
        thread.start()


        self.clients.append(client)
        print(f"Active connections: {len(self.clients)}")
```

Figure 12

The server accepts all incoming connections and receives their nicknames and a new car is created and stored in dictionary for each connected user alongside with its coordinates. If a car exists for a client that was previously connected the data is restored the that client and the new client connection is broadcasted to all the connected users. Finally, a thread is launched which is responsible for handling each client separately.

```
def handle_client(self, client, car_address):
    while True:
        try:
            #score ==> 1 coor ==> 2 ID ==> 0
            print("in try in client handle")
            # Receive the updated coordinates from the client
            data = client.recv(1024).decode()
            if data == "":
                raise Exception("empty message spam")

            splitted_data = data.split('$')
            for each_data in splitted_data:
                if each_data == '':
                    splitted_data.remove(each_data)

            print(data)
            print(splitted_data)
            print("check1")
            for each_data in splitted_data:
                coords = each_data.split("|")[2]
                car_x, car_y = self.read_pos(coords)# throw an exception here bc data is empty
                print("check2")
                car.car_x_coordinate = car_x
                car.car_y_coordinate = car_y
                print("check3")
                # Broadcast the updated coordinates to all clients
                self.serveravailable[address]= coords
                self.serverscore[address]= int(each_data.split("|")[1])
                print(self.serveravailable)
                print(self.serverscore)

                for c in self.clients:
                    if c != client:
                        #myData = str(address) + "|" + data
                        myData = each_data + "$"
                        c.sendall(myData.encode())
                    else:
                        print("check4")

        except Exception as e:
            # Handle client disconnection
            print("in exception in client handle")
            print(e)
            index = self.clients.index(client)
            self.clients.remove(client)
            car = self.cars[index]
            self.cars.remove(car)
            client.close()
            break
```

Figure 13

The handle function keeps receiving messages from the client the data is split first using the dollar sign which marks the end of the message and then it is split read the coordinates sent from the client and then it sent to all the other clients.

Backup:

The backup and migration feature was applied within the chat server

```
def handle_backup():
    main_server, address = server.accept()
    main_server.settimeout(3)
    while True:
        try:
            backup = main_server.recv(1024).decode('utf-8')
            if backup != "":
                print(backup)
            else:
                print("back up message is empty")
                raise Exception

        except:
            print("server crashed")
            try:
                main_server.close()
                receive()
            except:
                print("something happened while launching the backup server")
                break
```

Figure 14

When the connection between the backup server and the main server is interrupted or timed out the backup server will start accepting new client connections and it will start behaving as same as the main server. On the other hand, if the connection is aborted on the client side the clients will close the connection with the main server and will attempt a new connection with the backup server.

Hosting:

AWS EC2 was used to host the python server files on instances and run them from those instances.

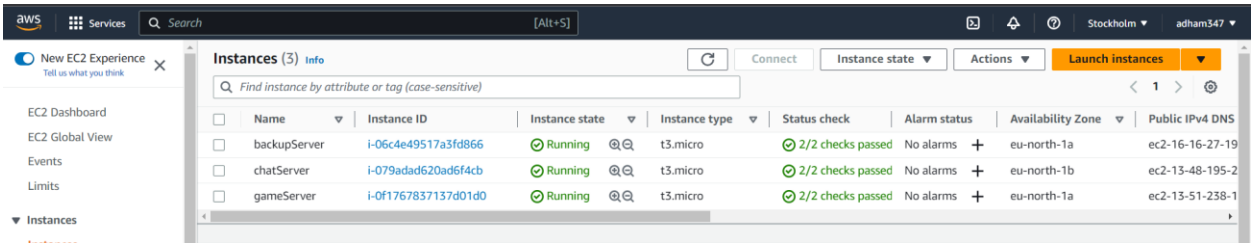

Figure 15

The files are uploaded using the SCP protocol to the instances created. And to connect to these instances and run the python scripts the SSH protocol is used.

```
D:\uni semester 8\CarRacing-DistributedProject>scp -i chatServerKey.pem server.py ec2-user@ec2-13-51-238-1.eu-north-1.co
mpute.amazonaws.com:/home/ec2-user/project
server.py                                     100% 4661    43.8KB/s   00:00
```

Figure 16

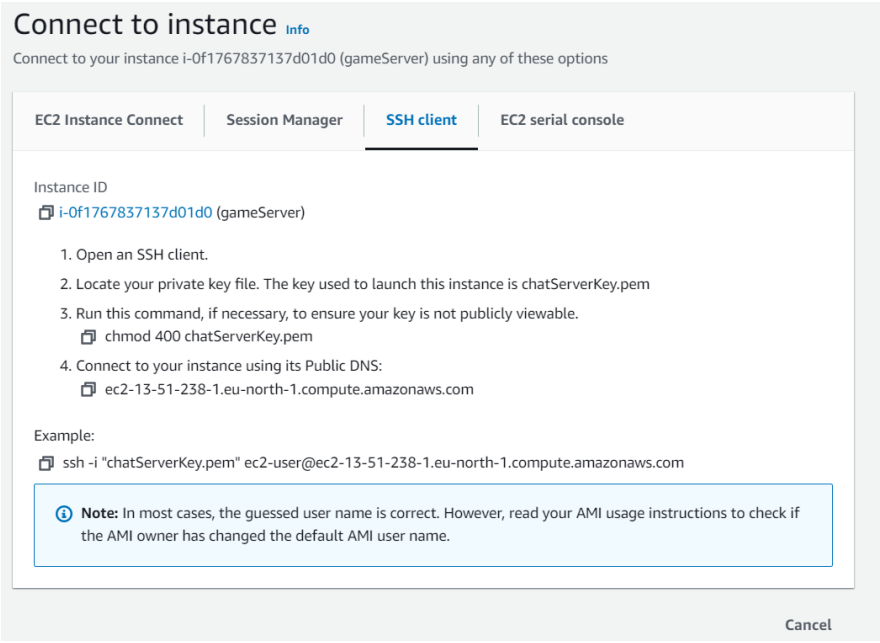

Figure 17

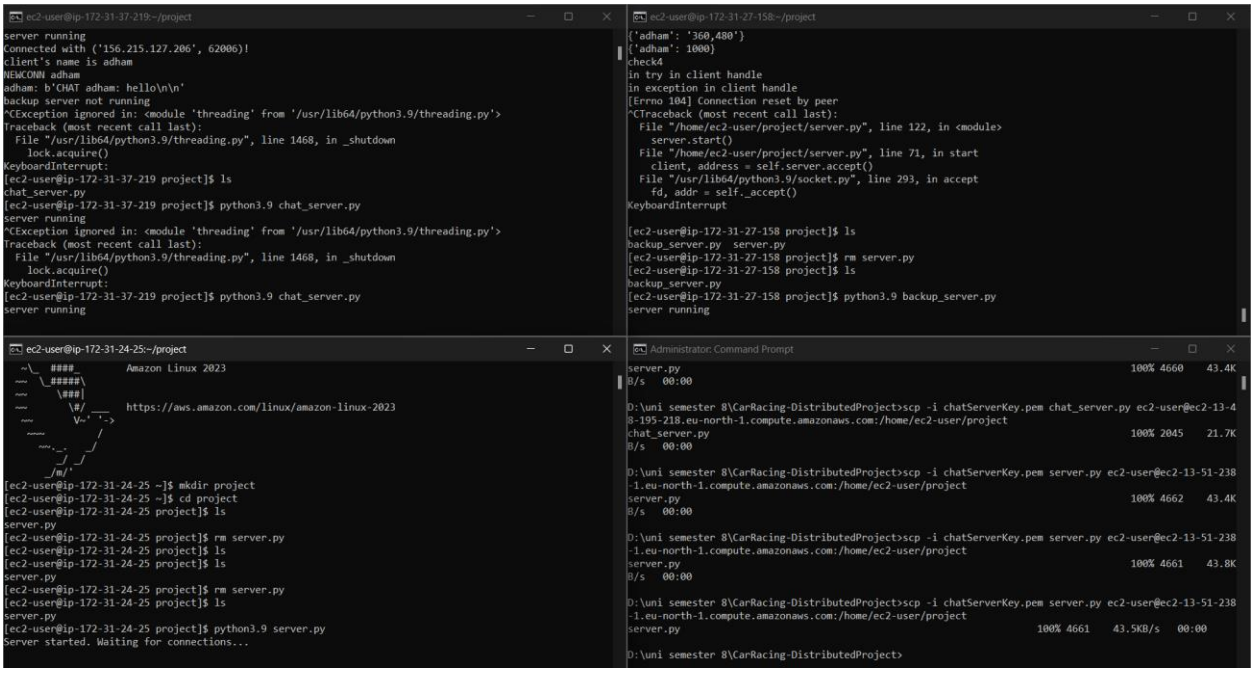

Figure 18

Now each machine has the python files on it and are up and running waiting for connections.

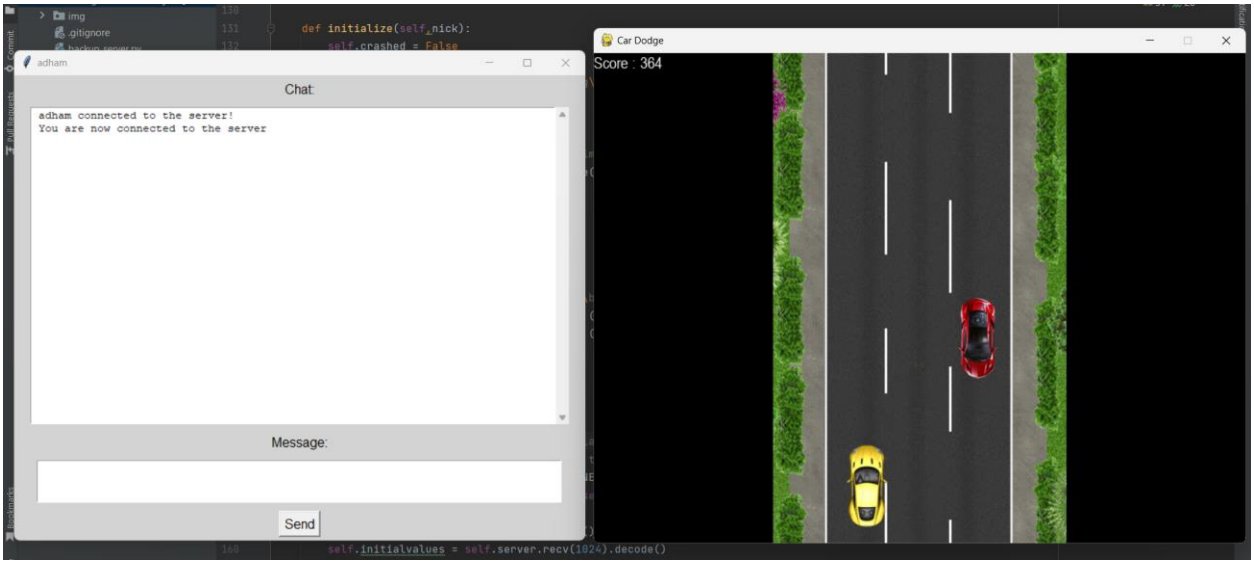

Figure 19

System Architecture and Design

The system is built over a basic socket programming and using a client-server architecture.

For each user there is one client program running and there are two servers one for the chat and another for the game.

End – User Guide:

The program doesn't have complications to start. First, you must make sure that the AWS instances are running, and the python scripts are being run on these instances. Then, just run cargame_T.py file. Afterwards, you will be asked for to enter a nickname and that's it the game will start and you will be able to play and chat successfully.