

Distributed Project



Program: CESS

Course Code:
CSE 354

Course Name:
Distributed Computing

Team 15

Mostafa Essam	18P9203
Noureldein khaled	18P5722
Madonna Magdy	18P2671
Mostafa Hesham	18P7502

Ain Shams University
Faculty of Engineering
Spring Semester – 2023

Dr. Ayman Bahaa
Ta. Mostafa Ashraf



Introduction:

This project involves developing a multiplayer distributed 2D Car Racing Game with a chat feature. The game allows multiple players, both human and AI-controlled agents, to compete in real-time on shared tracks. The game state is distributed across server nodes for scalability and fault tolerance.

Table of Contents:

- Project Description
- Beneficiaries
- Detailed Analysis
- Task Breakdown Structure
- Role of each member
- System architecture and design
- Testing scenarios
- End-User Guide
- Conclusion
- References
- Drive Link



Project Description:

The aim of this project is to develop a captivating multiplayer distributed 2D Car Racing Game with an integrated chat feature. The game will provide an engaging and competitive environment where players can race against each other in real-time, while also allowing them to communicate and interact through chat during, before, and after gameplay.

The game will support both human players and AI-controlled agents as opponents, ensuring a challenging experience for participants of varying skill levels. Players will have the opportunity to compete on dynamic tracks, utilizing power-ups and navigating through obstacles to gain an edge over their rivals. The game mechanics will include realistic car physics and precise collision detection to enhance the authenticity and excitement of the racing experience.

To ensure a robust and scalable system, a client-server architecture will be implemented. The server will serve as the central authority, managing car and block positions and facilitating communication between clients. The game state will be distributed across multiple server nodes to enhance performance and fault tolerance.

To support real-time playing and viewing by multiple participants, the game will incorporate mechanisms for real-time updates. This will ensure that all players have a synchronized view of the race and can experience smooth and responsive gameplay. The game's distributed nature will allow for many participants to join and compete simultaneously, fostering an immersive and dynamic multiplayer experience.

The chat feature will enable players to communicate with each other during the game, creating opportunities for strategic discussions, friendly banter, and community-building. Messages will be broadcast to all connected players in real-time, promoting a sense of camaraderie and competition.

To ensure the system's robustness, various measures will be implemented. Redundancy and replication techniques will be employed to safeguard the game state and prevent data loss in case of server failures. Load balancing mechanisms will distribute player connections across server nodes to maintain optimal performance and prevent overload. Error handling and recovery mechanisms will be in place to handle crashes and network failures, allowing the game to continue without major disruptions.

Beneficiaries:

The beneficiaries of the multiplayer distributed 2D Car Racing Game with a chat feature can be categorized into several groups:

Gamers:

Players looking for an immersive and competitive racing experience will benefit from the engaging gameplay, challenging tracks, and real-time multiplayer interactions.

Racing enthusiasts will enjoy the realistic car physics and the opportunity to showcase their racing skills against both human players and AI-controlled agents.



Game Developers:

Developers interested in creating multiplayer games can benefit from studying the project's client-server architecture, real-time updates, and distributed state management, which provide insights into building robust and scalable multiplayer systems.

Detailed Analysis:

Multiple Participants:

The game should support multiple participants, including both human players and autonomous agents. This requires designing a system that can handle a variable number of participants and provide a seamless experience for all players. The system should allow participants to join or leave the game dynamically and ensure fair competition between human and automated players.

Shared Resources and Real-time Updates:

In the game, participants will contend for shared resources, such as power-ups, tracks, or leaderboards. The system needs to handle real-time updates to the shared state to reflect changes in resource ownership, positions of players, and other relevant game events. Efficient data synchronization techniques, such as real-time data streaming or event-driven architectures, can be employed to ensure timely updates across all participants.



Robustness and Fault Tolerance:

The system should be robust and able to handle failures gracefully. If a participant node crashes, the game should continue without disruption for other participants. To achieve this, redundancy measures such as data replication, failover mechanisms, or decentralized architectures can be implemented.

Real-time Gameplay and Viewing:

The game should support real-time playing and viewing by multiple participants. This requires a responsive and low-latency system to ensure that all participants experience smooth gameplay. Techniques such as client-side prediction, server interpolation, and network optimizations can be employed to minimize lag and provide a synchronized experience for all players.

Chatting Feature:

Implementing a chat feature enables participants to communicate with each other during/before/after playing the game. The chat system should support real-time messaging, with features like private messaging, chat rooms, and message moderation if required. Security measures should be implemented to prevent abuse and ensure a positive and inclusive gaming environment.



Task Breakdown Structure:

Server file:

Responsible for creating a server for the game and maintaining communication with the clients. It creates a thread for each client that connects to it and starts to send and receive messages and information through the threads.

Client file:

The client functions are implemented in this file then later on are moved to the main.py

Main file:

This is the main file of the project where the game is made. The main menu screen is opened first and a connection with the server is established, and an ID is given to each client in the order they connected. The Client can either choose to Quit or Start the game. When start is pressed the `game_loop()` function is called and the game begins.

The player can move his car using the arrow keys and check for collisions with blocks, if a collision has occurred the score is decreased and a message is sent to the server to determine the new position of the blocks, and if no collision happens and the blocks reach the end of the screen a message is sent to the server to determine the new position of the blocks.



Database file:

This is the initial version of the database connection is made in this file. The functions and connections in this file are later transferred to the server.

Server1chat file:

This file is the chat server. It is responsible for creating a room which clients can connect to and message each other. The clients can choose an alias which is shown when they connect and when they send a message. If a client leaves the chat a message is displayed to notify the others.

clientchat file:

This file is the chat client. It is responsible for sending and receiving messages to the server.



Role of each member:

Mostafa Essam -> Game concept – Gui – Integration of the game with the server and the client

Madonna Magdy -> server1 – server2 – client - chatServer – integration of the game with the server and the client

Mostafa Hesham -> MongoDB Tables creation – Data Base code – integration of the database with server

Noureldien Khaled -> client of chat server – deployment – helped with game.

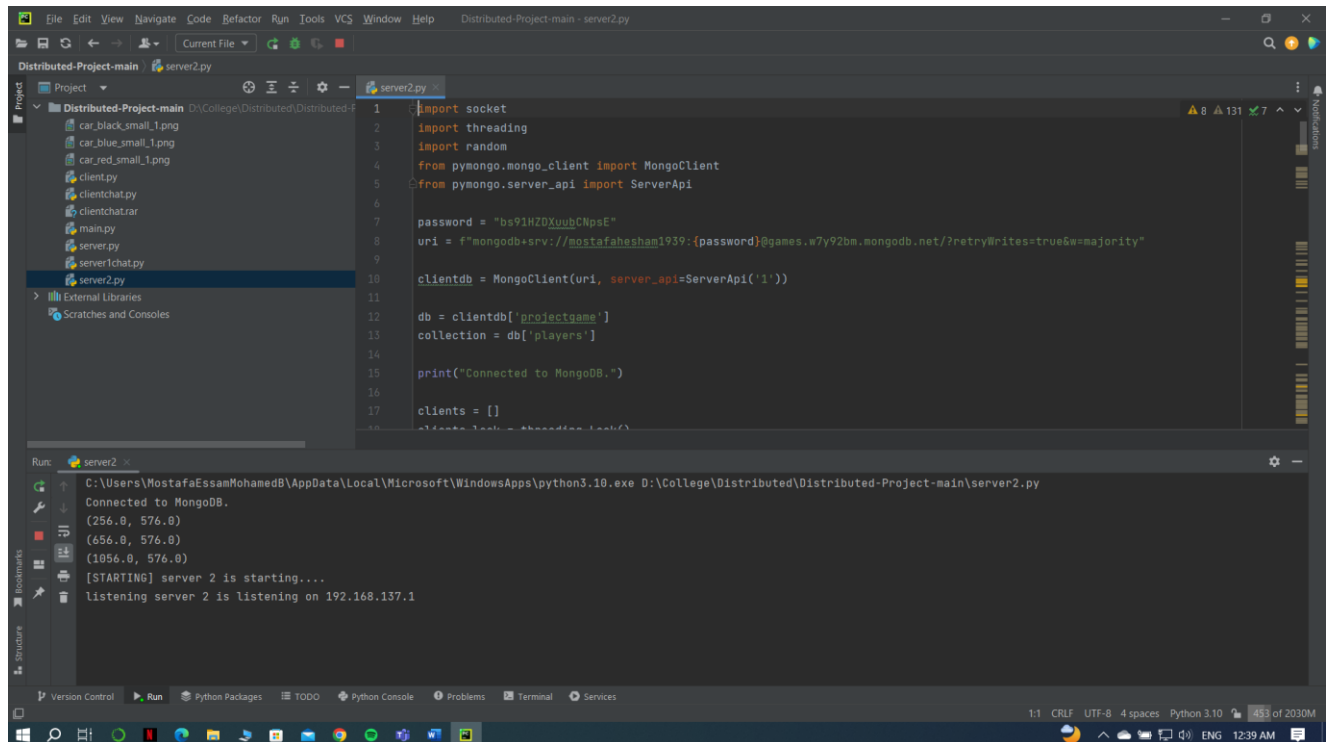
System architecture and design:

Client-Server Game Architecture: The system can employ a client-server architecture, where clients (participants) interact with a central server responsible for managing the game state, processing inputs, and facilitating communication. The server can handle synchronization of game state across clients.

Client-Server Chat Architecture: the system supports a chat function that allows users to communicate via messages.

Testing scenarios:

1- We start the server and wait for connection

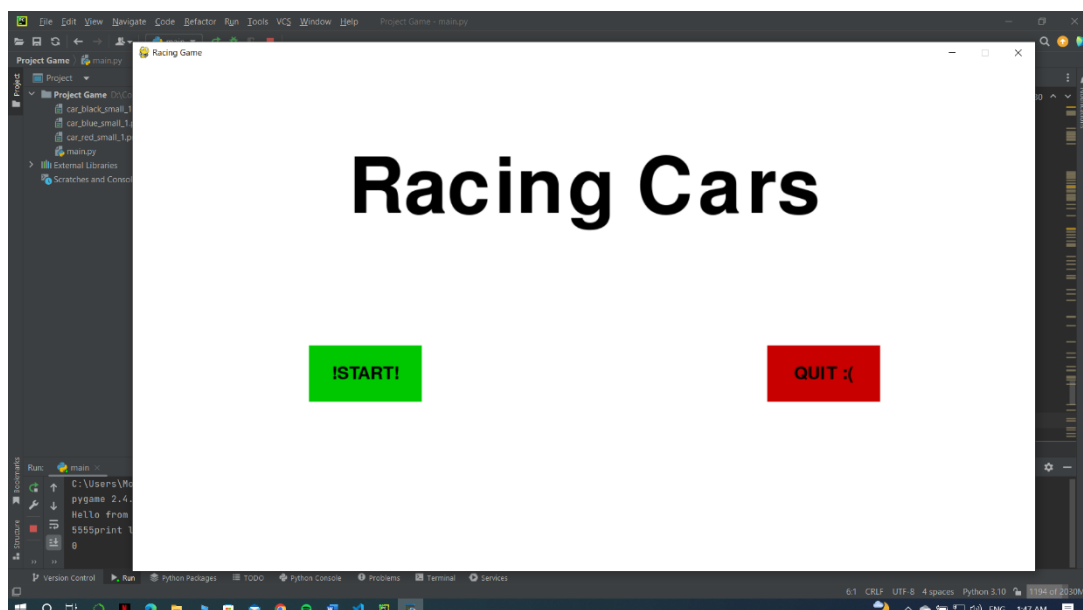


```
1 import socket
2 import threading
3 import random
4 from pymongo.mongo_client import MongoClient
5 from pymongo.server_api import ServerApi
6
7 password = "bs91H2DxubCMpsE"
8 uri = f"mongodb+srv://{goatfahesham1939:{password}@games.w7y92bm.mongodb.net/?retryWrites=true&w=majority"
9
10 clientdb = MongoClient(uri, server_api=ServerApi('1'))
11
12 db = clientdb['projectgame']
13 collection = db['players']
14
15 print("Connected to MongoDB.")
16
17 clients = []
18 clients_lock = threading.Lock()
```

Run: server2

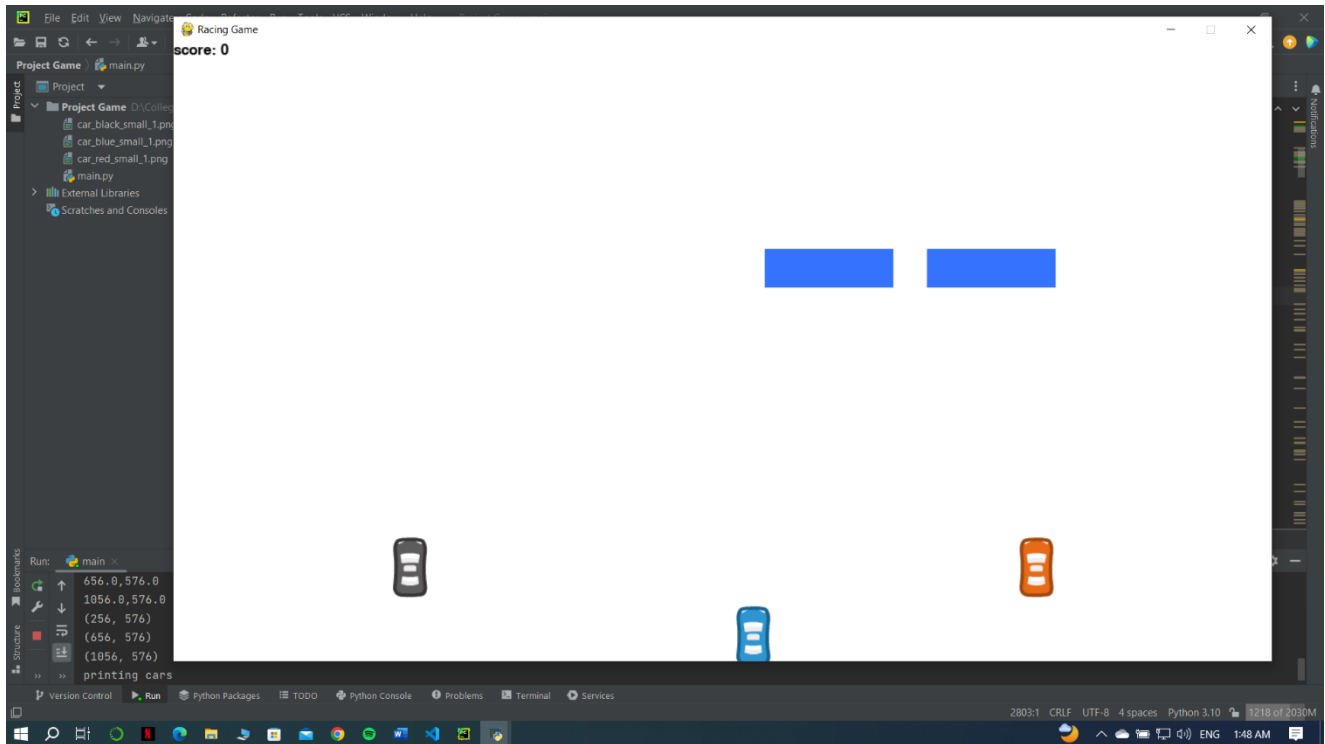
```
C:\Users\MostafaEssamMohamedB\AppData\Local\Microsoft\WindowsApps\python3.10.exe D:\College\Distributed\ Distributed-Project-main\server2.py
Connected to MongoDB.
(256.0, 576.0)
(656.0, 576.0)
(1056.0, 576.0)
[STARTING] server 2 is starting....
listening server 2 is listening on 192.168.137.1
```

2- We start the game and get an ID

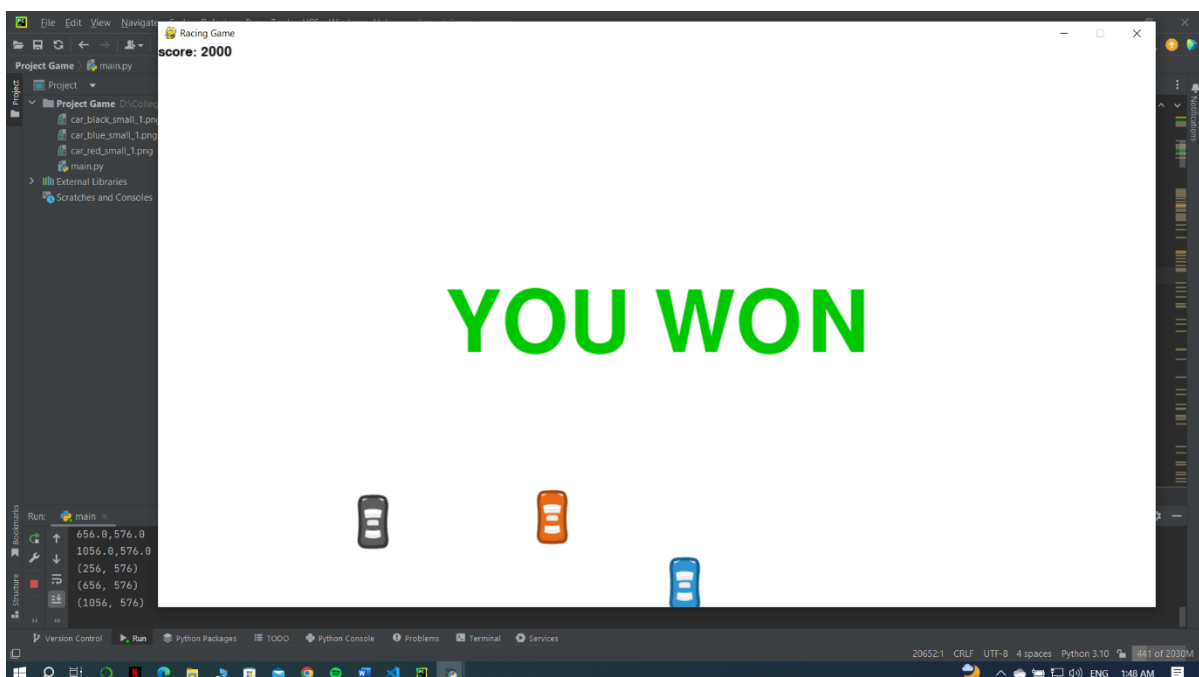


3- We can press Quit and exit the game

4- We press Start and start the race



5- When the client reaches the max score the game ends and a message is displayed on the screen





User End Guide:

- 1- check which server flag is up main.py and run the corresponding server
- 2- run the main.p file
- 3- press START to join the game and play
- 4- Press Quit to EXIT

Conclusion:

In conclusion, the development of a multiplayer distributed 2D Car Racing Game with a chat feature involves various components and considerations. The system architecture and design play a crucial role in ensuring the robustness, scalability, and real-time capabilities of the game.

By employing a client-server architecture, distributing the game state across multiple server nodes, and utilizing real-time communication protocols like Sockets, the system can provide a seamless multiplayer experience. The implementation of a game logic and physics engine, participant management functionalities, crash recovery mechanisms, and a chat system further enhance the gameplay and social interaction aspects.



References & Links:

<https://www.youtube.com/watch?v=nmzzeAvQHp8>

https://www.youtube.com/playlist?list=PLQVvvaa0QuDdLkP8MrOXLe_rKuf6r80KO

Drive Link:

<https://drive.google.com/drive/folders/1VPwb7Zm5EpxvUFVG8sCgMP6g1Ib5-uOK?usp=sharing>