CSE354

Distributed Computing

Distributed Project

A multi-player distributed 2D Car Racing Game along with a chatting feature

Team 8

Name	**ID**
Seifeldin Sameh Mostafa Elkholy	19p3954
Yara Mostafa Ibrahim	19p1120
Mohamed Amr Fathy Nasef	20p5552
Omar Mohamed Shams Eldeen	21p0188

Table of Contents

Table of Figures

Introduction

Welcome to the exciting world of a distributed car game with a chatting feature! This game allows multiple players to race against each other in real-time, while also being able to communicate through the chat feature, you can also create multiple rooms for your group. With good graphics and realistic car physics, players can experience the thrill of competing during the race score is calculated and a leader board is shown to each player, all while interacting with other players.

Task breakdown

1) Chat GUI
2) Creating chat server logic, creating a multithreaded server to receive multiple clients
3) Modifying car game to make it eligible to add multiple players, and creating a multithreaded server to receive multiple players
4) Database for the system to save room name, player name and their score and additional info and connecting code to database.
5) Integration of whole system
6) Hosting on AWS and testing
7) Documentation

Roles in each task

Seif: 1,2,3,4,5,7

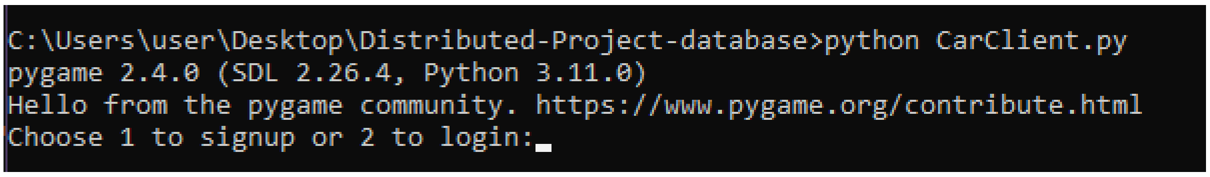
Yara: 1,2,3,4,5,7

Omar: 4,5

Nassef: 6

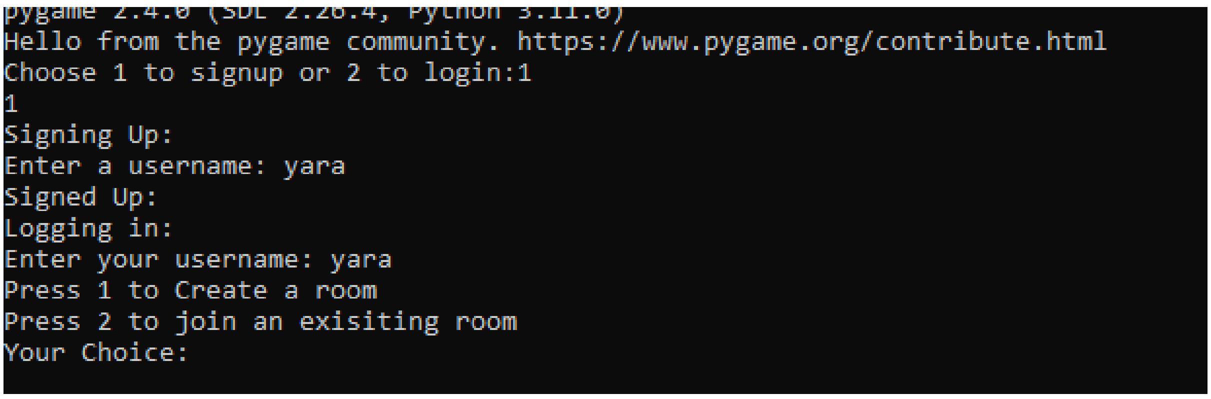
Detailed project description

We created a multiplayer distributed car game with chatting feature, we used modules such as socket, threading, pickle, and DB, to support the distributed muti-player option, each player joins must either log in or sign up if their data is saved before on database then the player can log in if not then he has to write his name and join or create a room and choose the number of players they want this room to have. Each username is unique.



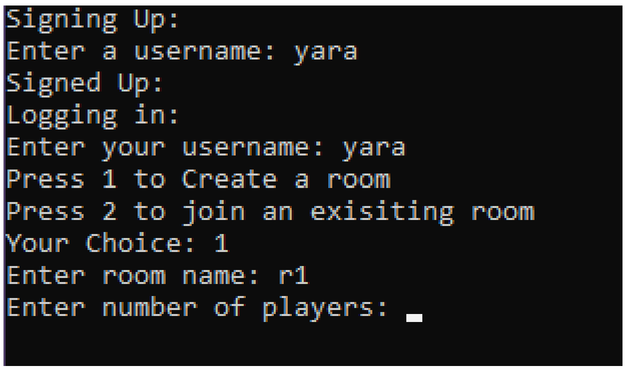
```
C:\Users\user\Desktop\Distributed-Project-database>python CarClient.py
pygame 2.4.0 (SDL 2.26.4, Python 3.11.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
Choose 1 to signup or 2 to login:_
```

demo 1



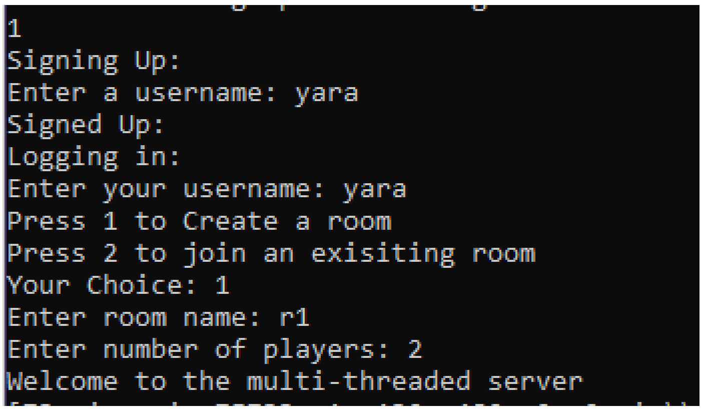
```
pygame 2.4.0 (SDL 2.26.4, Python 3.11.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
Choose 1 to signup or 2 to login:1
1
Signing Up:
Enter a username: yara
Signed Up:
Logging in:
Enter your username: yara
Press 1 to Create a room
Press 2 to join an exisiting room
Your Choice:
```

demo 2



```
Signing Up:
Enter a username: yara
Signed Up:
Logging in:
Enter your username: yara
Press 1 to Create a room
Press 2 to join an exisiting room
Your Choice: 1
Enter room name: r1
Enter number of players: _
```

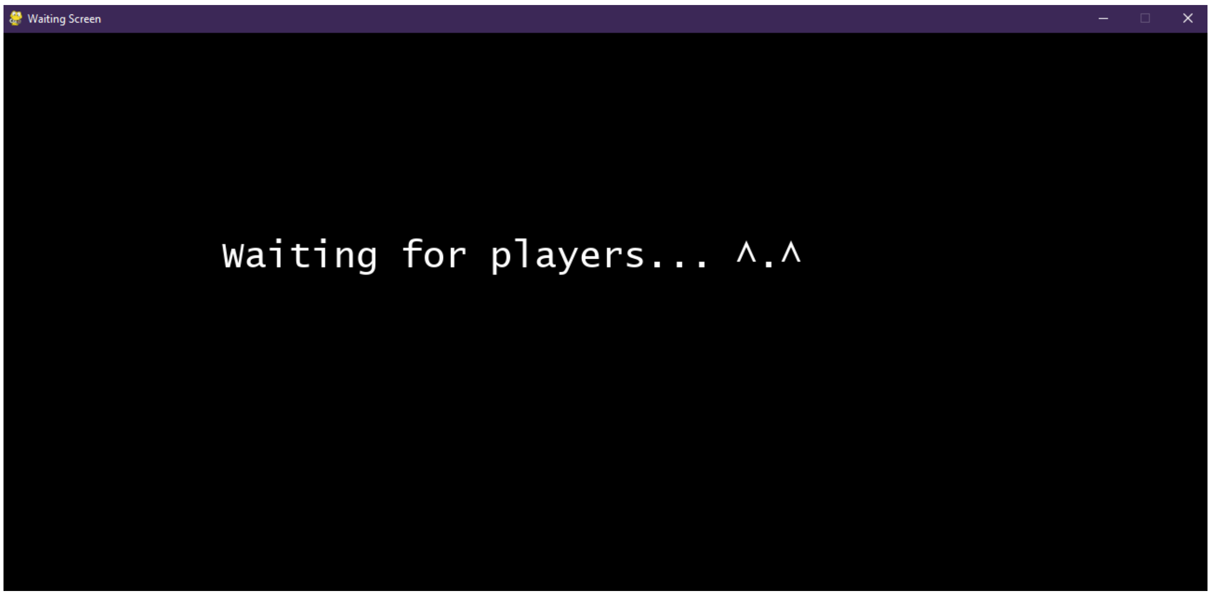
demo 3



```
1
Signing Up:
Enter a username: yara
Signed Up:
Logging in:
Enter your username: yara
Press 1 to Create a room
Press 2 to join an exisiting room
Your Choice: 1
Enter room name: r1
Enter number of players: 2
Welcome to the multi-threaded server
```

demo 4

A loading screen is shown to players until all players join the room.

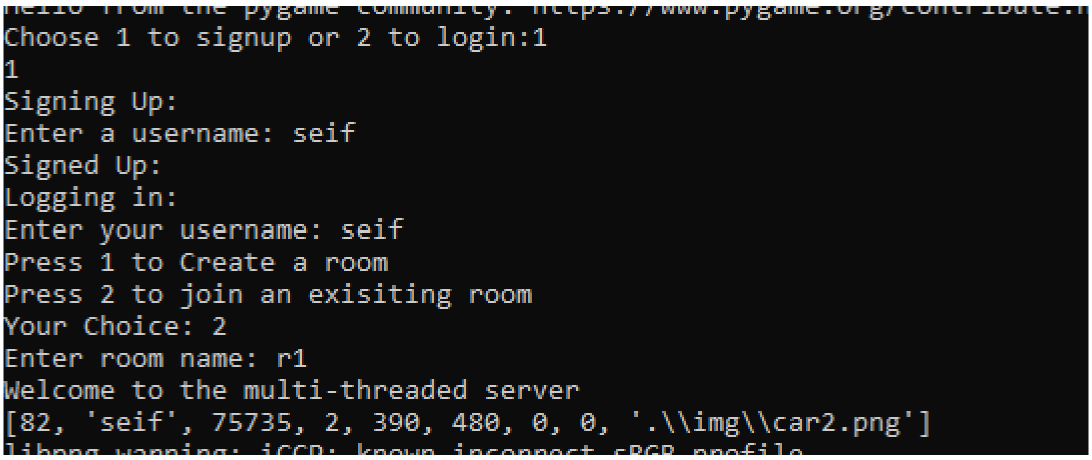


Waiting Screen

Waiting for players... ^.^

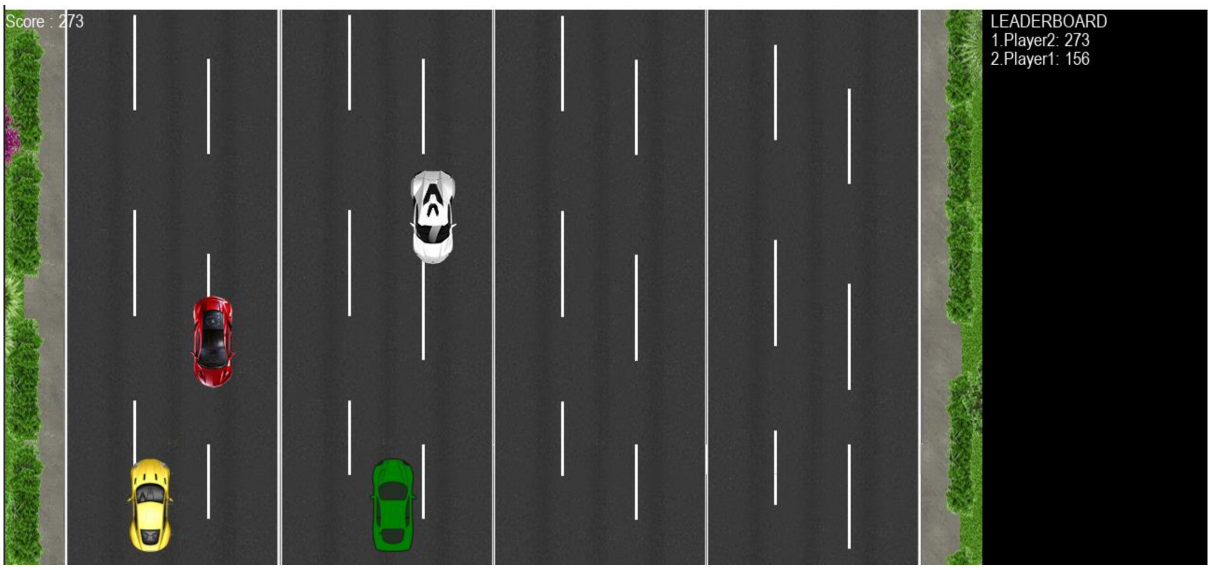
demo 5

After the number of players specified joins the room the game starts.



```
Choose 1 to signup or 2 to login:1
1
Signing Up:
Enter a username: seif
Signed Up:
Logging in:
Enter your username: seif
Press 1 to Create a room
Press 2 to join an exisiting room
Your Choice: 2
Enter room name: r1
Welcome to the multi-threaded server
[82, 'seif', 75735, 2, 390, 480, 0, 0, '.\\img\\car2.png']
```

demo 6



demo 7

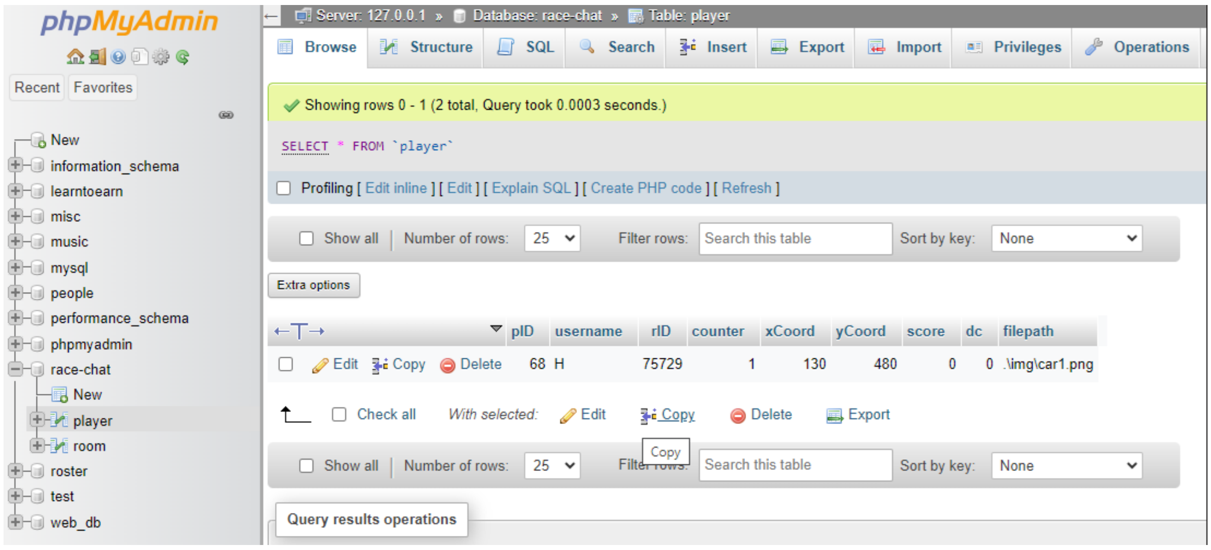
A leader board is shown on the right for each player and is ordered according to scores.

Once user connects to the server, the user joins a chat room across all car rooms.

The server notifies the rest of users when a user joins and when they disconnect.

Users are free to continue sending even after the game ends.

Our database "race-chat.sql" contains two tables that save the information of each player.

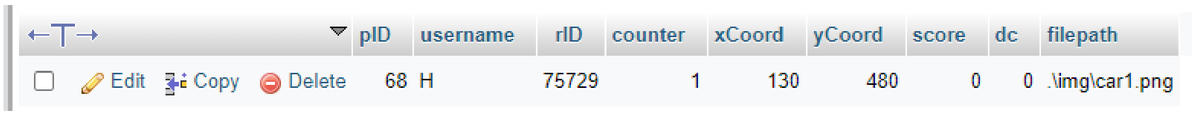


Showing rows 0 - 1 (2 total, Query took 0.0003 seconds.)

```
SELECT * FROM `player`
```

	pID	username	rID	counter	xCoord	yCoord	score	dc	filepath
Edit Copy Delete	68	H	75729	1	130	480	0	0	.\img\car1.png

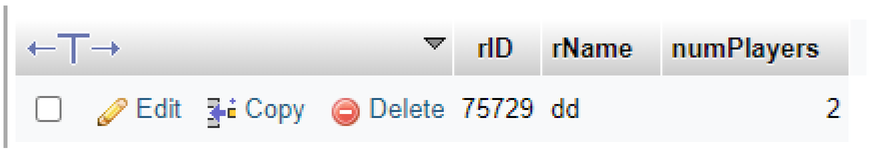
In table player



	pID	username	rID	counter	xCoord	yCoord	score	dc	filepath
Edit Copy Delete	68	H	75729	1	130	480	0	0	.\img\car1.png

Username, position and score of the player is saved, in case user disconnect

Table room



	rID	rName	numPlayers
Edit Copy Delete	75729	dd	2

The room name is saved, and they number of players that ought to join the room for the game to begin.

Beneficiaries of the project

- Gaining experience in designing and implementing complex distributed systems, as well as learning about PyGame and implementing it in a distributed manner alongside chatting features.
- Handling multiple corner cases and catching exceptions
- Marshaling and Un-marshaling objects and data across servers and clients
- Implementing Client-Server model using sockets and socket-APIs
- Implementing threading using a dispatcher.
- Working with sockets to make a distributed system to provide scalability, flexibility, reliability, performance, and security.
- Working with AWS, rds and EC2 instances
- Writing system architecture using terraform

System Architecture

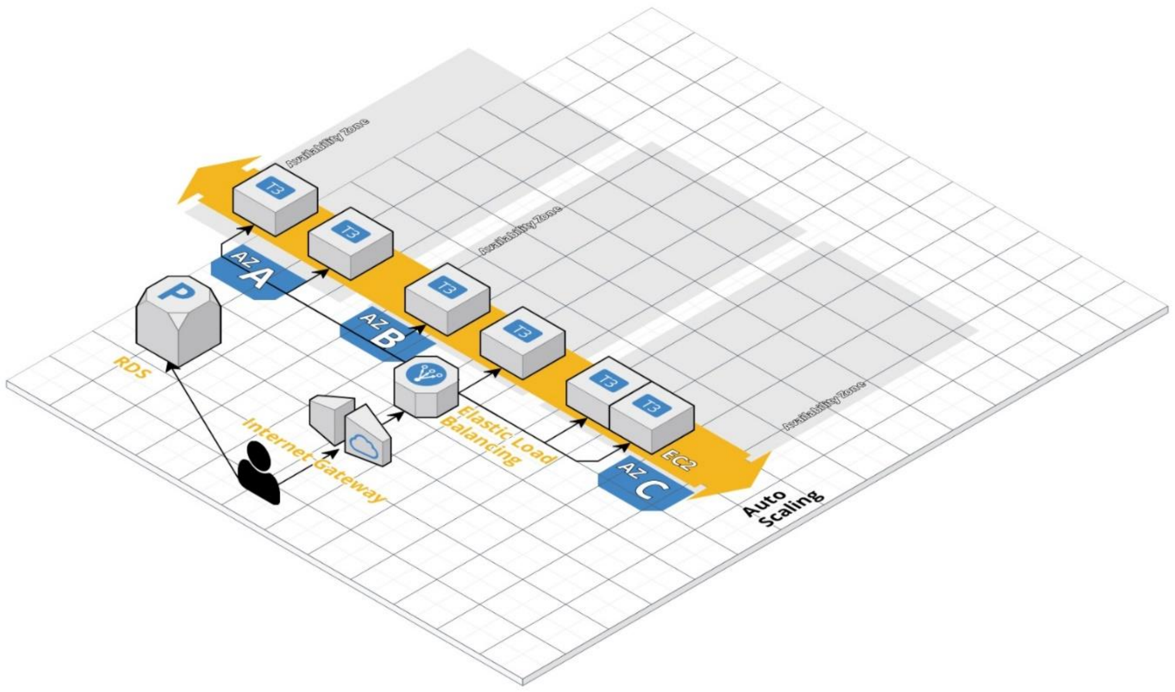


Figure 1

Researching best approach to system architecture

Our goal was to make our system architecture supports scalability, and fault tolerant.

Our first approach was to try Docker swarm.

Docker Swarm is a container orchestration tool provided by Docker, which allows us to create and manage a cluster of Docker hosts. It enables us to deploy and manage a distributed application across multiple machines, providing fault tolerance, scalability, and load balancing.

It has multiple useful properties:

1- Swarm Mode: Docker Swarm introduces the concept of Swarm Mode, which enables you to create and manage a swarm of Docker nodes. A swarm consists of one or more manager nodes and worker nodes. The manager nodes handle orchestration tasks, such as scheduling containers, maintaining the desired state, and managing the swarm cluster. The worker nodes execute the tasks assigned to them by the manager nodes.[1][2]

2- Service Abstraction: Docker Swarm uses the service abstraction to define the desired state of your application. A service is a long-running container that performs a specific task or provides a specific service. You can define the number of replicas (instances) of a service, the resources it requires, network configurations, and other parameters. Swarm ensures that the specified number of replicas are running across the swarm cluster, automatically handling rescheduling in case of failures.[1][3]

3- Load Balancing: Swarm provides built-in load balancing for services. When you create a service, a virtual IP (VIP) is assigned to it, acting as the entry point for accessing the service. Swarm automatically distributes incoming requests across the available replicas, spreading the load and ensuring high availability.[1][4]

4- Scaling: Docker Swarm allows you to scale your services up or down based on demand. You can increase the number of replicas for a service to handle higher traffic or decrease it during periods of low demand. Swarm takes care of distributing the workload across the available nodes and maintaining the desired state.[1][5]

5- Service Discovery: Swarm provides built-in service discovery, allowing services to find and communicate with each other by their service names. Each service is automatically registered with an embedded DNS server, enabling easy inter-service communication without hardcoding IP addresses or managing external service discovery mechanisms.[1][6]

6- Rolling Updates: Swarm supports rolling updates for services, enabling you to update your application without downtime. You can define update strategies, such as the maximum number of concurrent updates, the delay between updates, or the number of healthy replicas required before moving on to the next update. Swarm ensures that your application remains available while gradually updating the containers.[1][7]
7- Secrets Management: Swarm provides a secure way to manage sensitive information, such as passwords or API keys, using secrets. Secrets are encrypted pieces of data that can be provided to services during runtime. Swarm handles the distribution and encryption of secrets, making them available to the necessary services while keeping them secure. [1][8]

Docker turned out not to be the best fit to our project as it does not support auto-scaling.

Our next option was to seek out Kubernetes.

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes provides a robust framework for managing distributed systems and simplifies the management of containerized applications at scale.

Kubernetes properties includes:

1- Container Orchestration: Kubernetes acts as a container orchestration platform, enabling you to manage and coordinate multiple containers across a cluster of machines. It automates various tasks, such as container deployment, scaling, load balancing, networking, and scaling applications, making it easier to manage complex containerized environments.[8]

2- Cluster Architecture: Kubernetes follows a master-worker architecture. The master node controls the cluster and manages its state, while the worker nodes (also known as minions) execute the containers and applications. The master node provides an API server for managing the cluster, a scheduler for assigning work to the nodes, and a controller manager for maintaining the desired state of the cluster.[9]

3- Pod Abstraction: In Kubernetes, the basic unit of deployment is a pod. A pod is a logical group of one or more containers that share the same network namespace and storage resources. Containers within a pod can communicate with each other using localhost, simplifying inter-container communication. Pods are the atomic scheduling unit in Kubernetes and can be scaled up or down based on resource requirements.[10]

4- Service Discovery and Load Balancing: Kubernetes provides built-in service discovery and load balancing mechanisms. Services define a set of pods and provide a stable network endpoint for accessing them. Kubernetes automatically assigns a virtual IP (VIP) to each service, allowing other services or external clients to communicate with the pods. It load balances incoming requests across the available pods, ensuring high availability and efficient resource utilization.[11]

5- Scaling and Self-Healing: Kubernetes enables horizontal scaling of applications. You can define the desired number of replicas for a pod or a set of pods, and Kubernetes automatically scales the deployment based on resource utilization or custom metrics. Kubernetes also supports self-healing by monitoring the health of pods. If a pod fails or becomes unresponsive, Kubernetes automatically restarts or replaces it to maintain the desired state.[12]

6- Configuration and Secrets Management: Kubernetes provides mechanisms for managing application configuration and secrets. ConfigMaps allow you to store configuration data as key-value pairs or as files, which can be mounted into pods as volumes or environment variables. Secrets are used to store sensitive information, such as passwords or API keys, in an encrypted manner. They can be securely provided to pods without exposing the sensitive data directly.[13]

7- Rolling Updates and Rollbacks: Kubernetes supports rolling updates and rollbacks of applications, enabling seamless updates without downtime. You can define update strategies, such as the maximum number of simultaneous pods to update, the delay between updates, or health checks to ensure the new version is functioning correctly. If an update causes issues, Kubernetes allows you to rollback to the previous stable version.[14]

Kubernetes turned out not to be the best fit to our project as it cannot maintain a consistent copy of our database across multiple containers.

Finally, we settled on a Primary-Secondary Database scheme as it best fits out demands and needs.

This scheme would be developed on aws using managed instances groups services that supports auto-scaling and database consistency.

System Architecture Description

Our database is managed by Amazon Relational Database Service.

Clients access our servers through Internet Gateway.

Internet Gateway then directs traffic to an Elastic Load Balancer that balances traffic between EC2 instances.

EC2 instances auto-scale according to stress and traffic and is maintained automatically by aws.

Detailed analysis

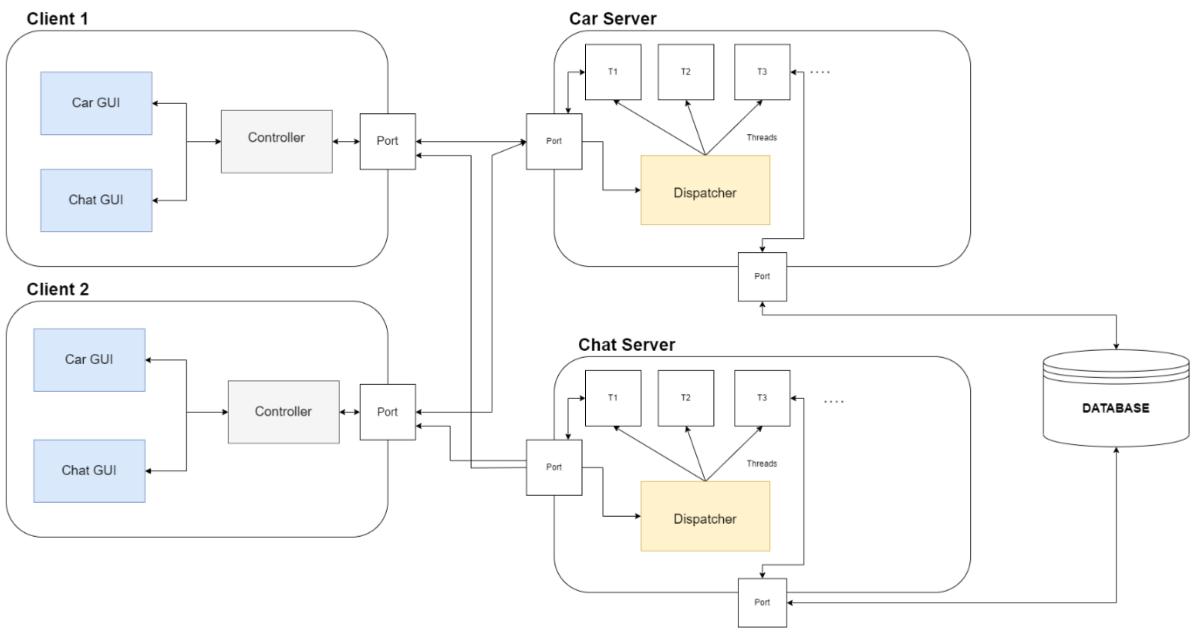


Figure 2

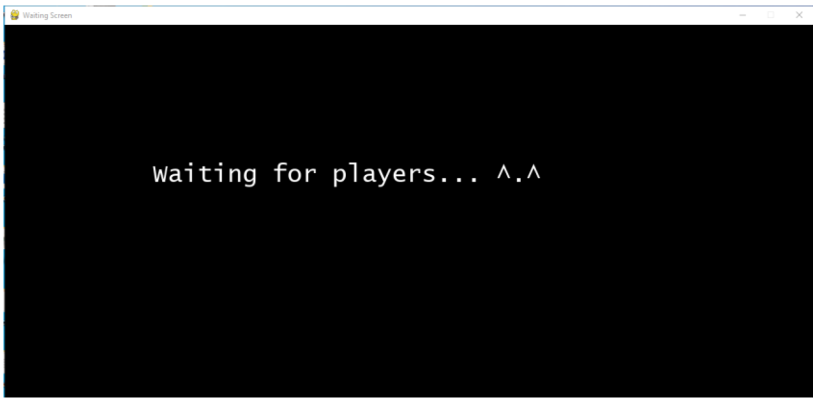
Client Side

Client program is divided into 2 main parts: 1-GUI, 2- controller.

GUI involves 2 GUIs, one for car games and other for chatting interface.

Car GUI

Waiting Screen:

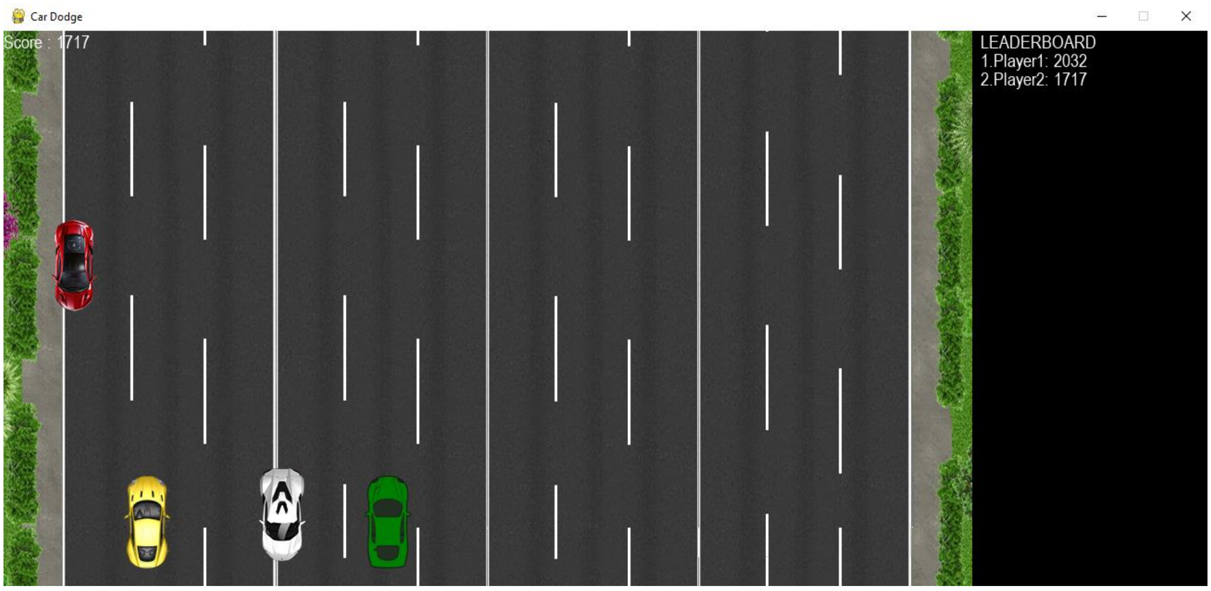


```
Waiting for players... ^.^
```

Figure 3

Waiting Screen is shown initially when the room is empty or still haven't got required number of players.

Two-Players room GUI:



```
Score : 1717
```

```
LEADERBOARD
1.Player1: 2032
2.Player2: 1717
```

Figure 4

The score for each player is shown to the other player on the leaderboard.

The top score is shown first then other scores are shown in a descending order.

In case one of the players disconnects, the other player is notified.

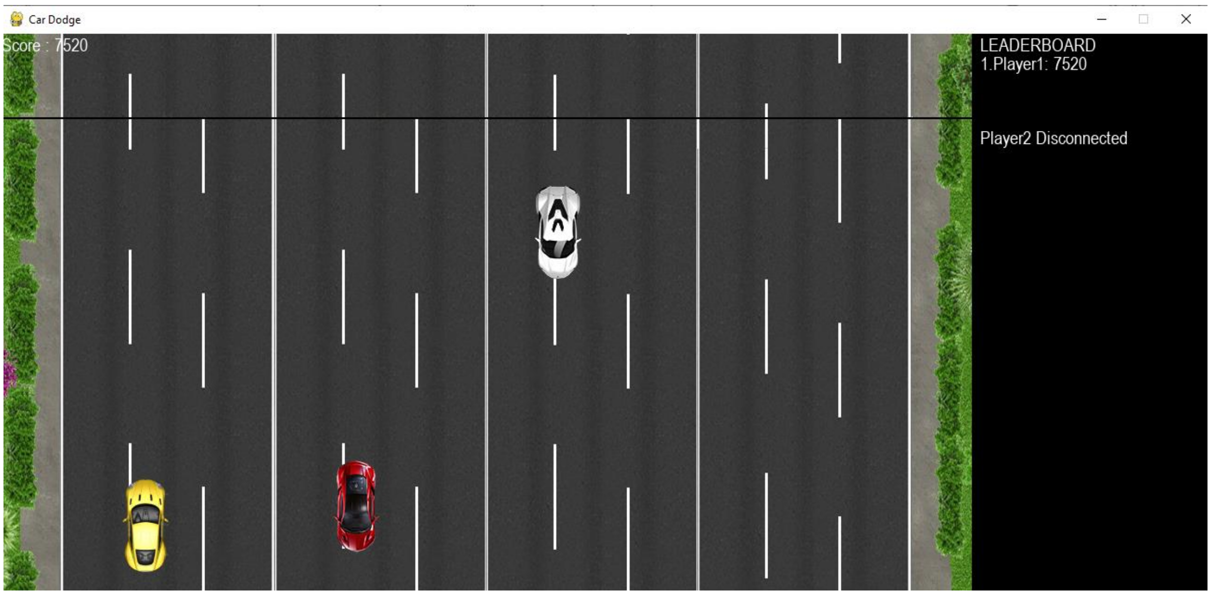


Figure 5

Three-Players room GUI:

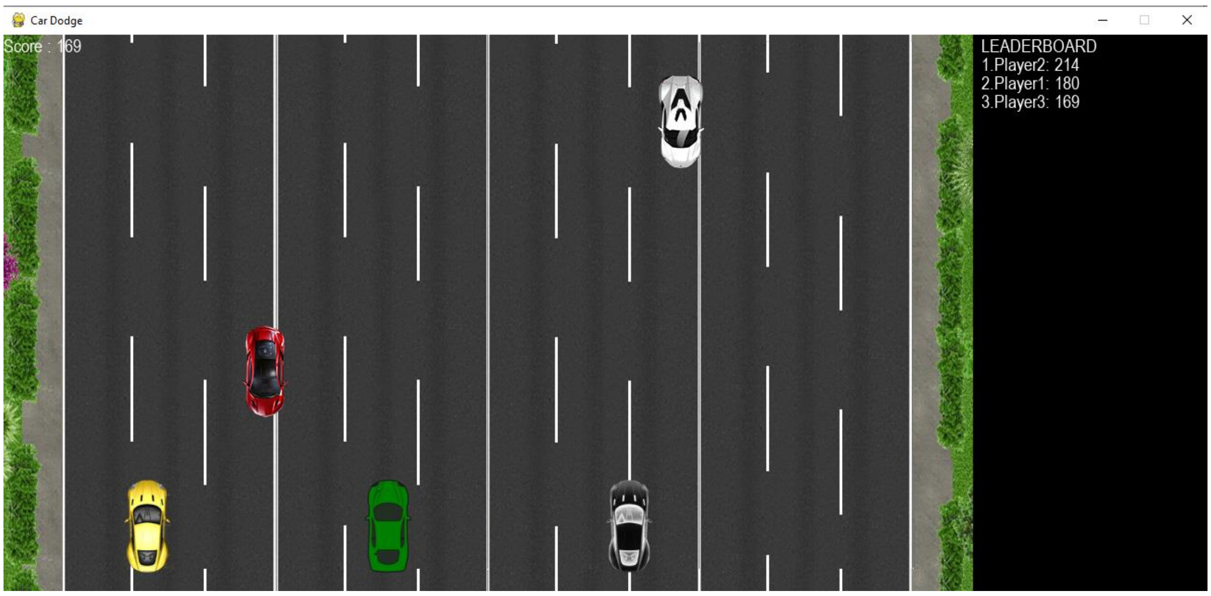
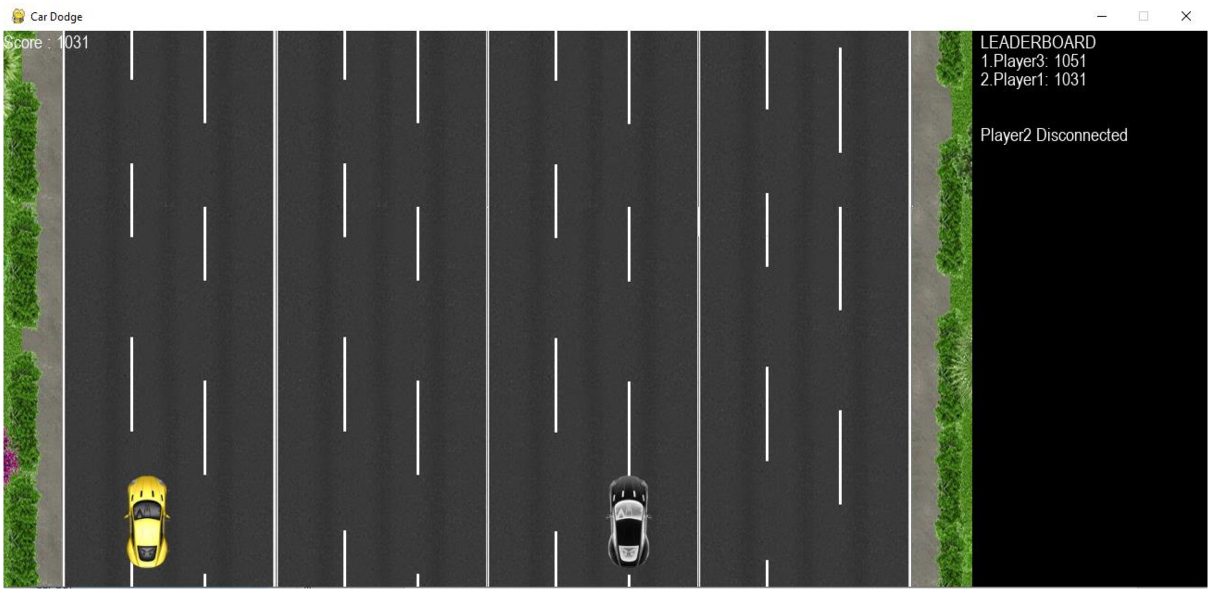


Figure 6

All players are notified if a player disconnects.

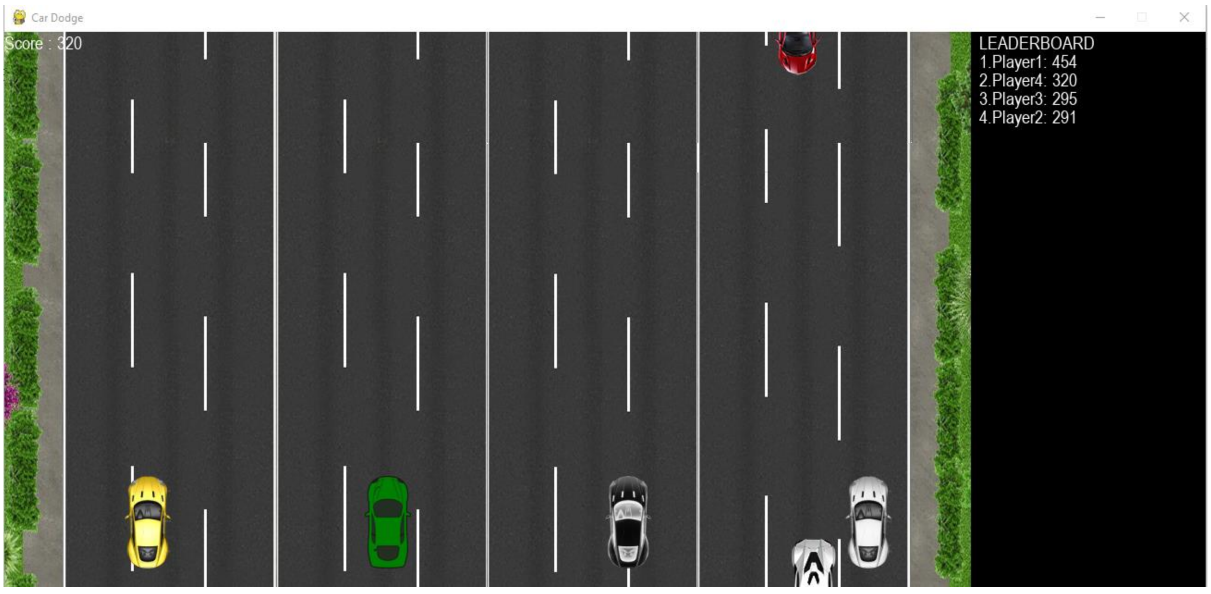


LEADERBOARD
1.Player3: 1051
2.Player1: 1031

Player2 Disconnected

Figure 7

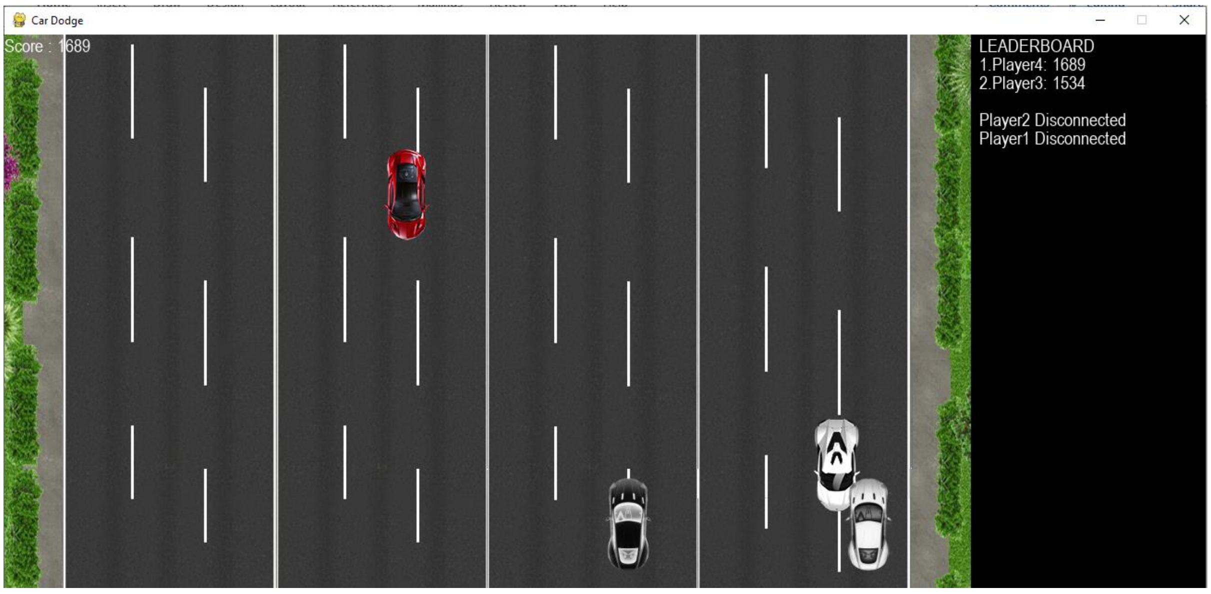
Four-Players room GUI:



LEADERBOARD
1.Player1: 454
2.Player4: 320
3.Player3: 295
4.Player2: 291

Figure 8

All other players are notified if a player disconnects.



LEADERBOARD
1.Player4: 1689
2.Player3: 1534

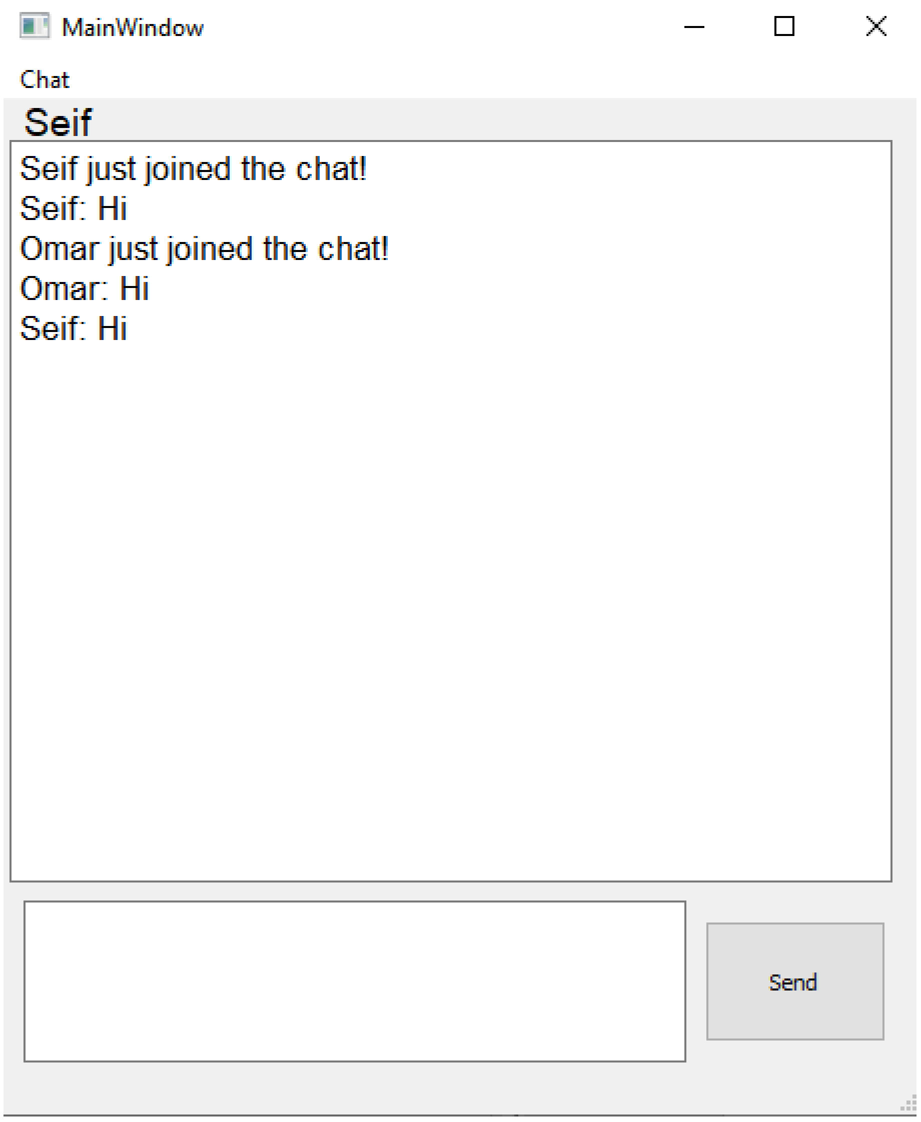
Player2 Disconnected
Player1 Disconnected

Figure 9

Chat GUI

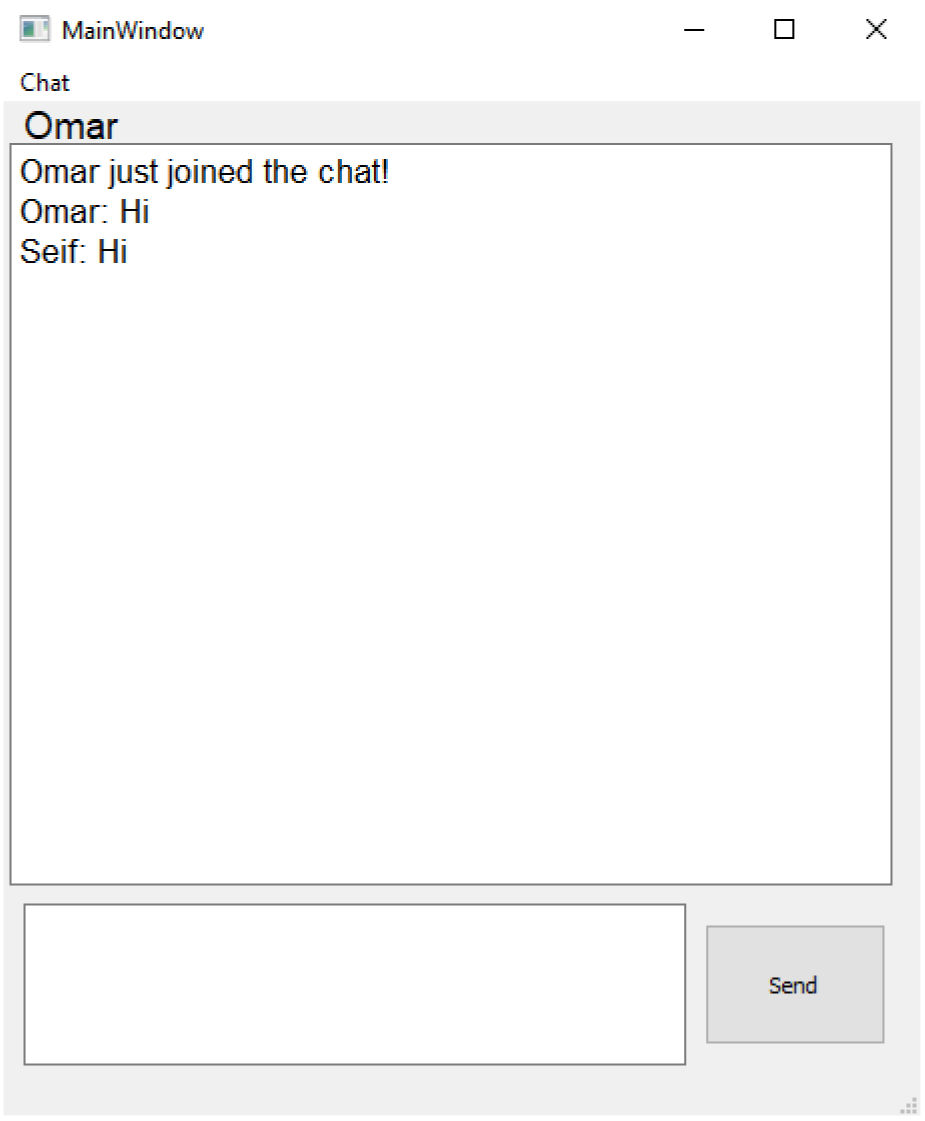
All users can chat in a single room across multiple games.

Users are notified when a new user joins the chat room.



Seif just joined the chat!
Seif: Hi
Omar just joined the chat!
Omar: Hi
Seif: Hi

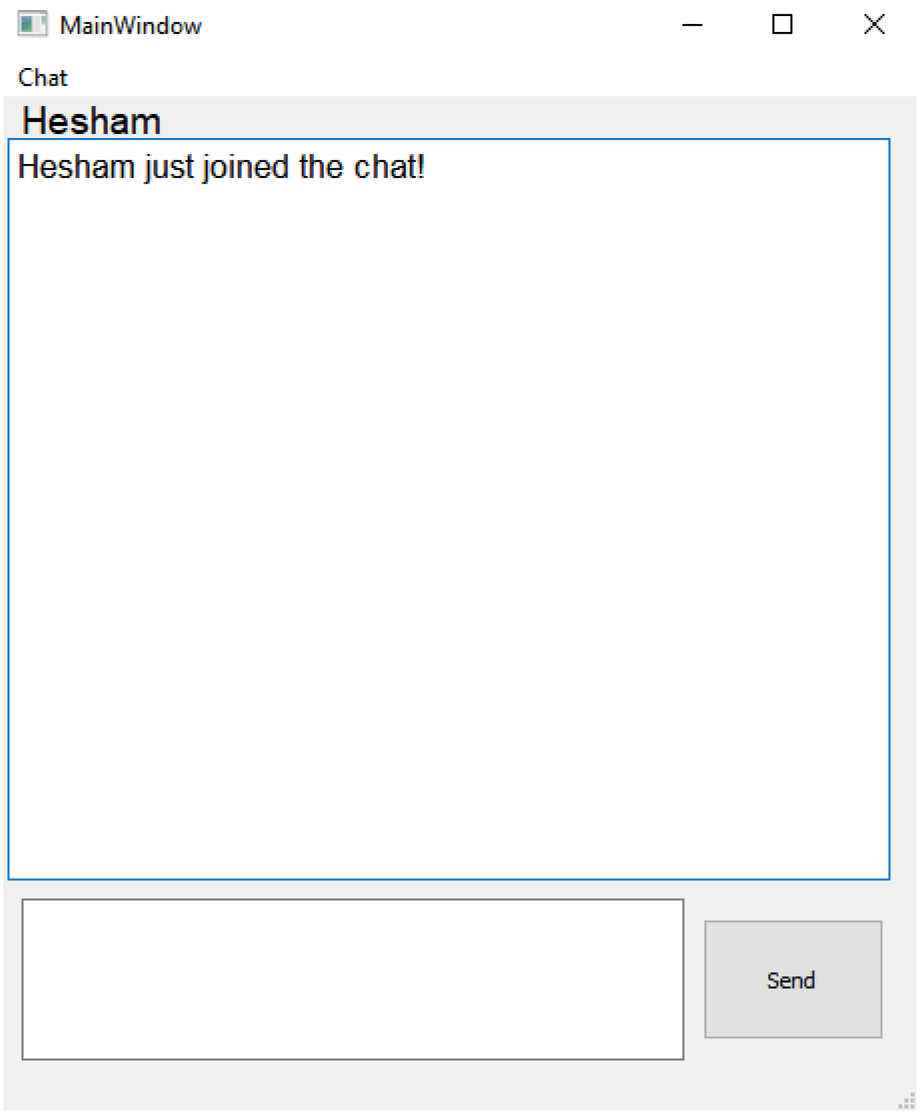
Figure 10



Omar

Omar just joined the chat!
Omar: Hi
Seif: Hi

Figure 11



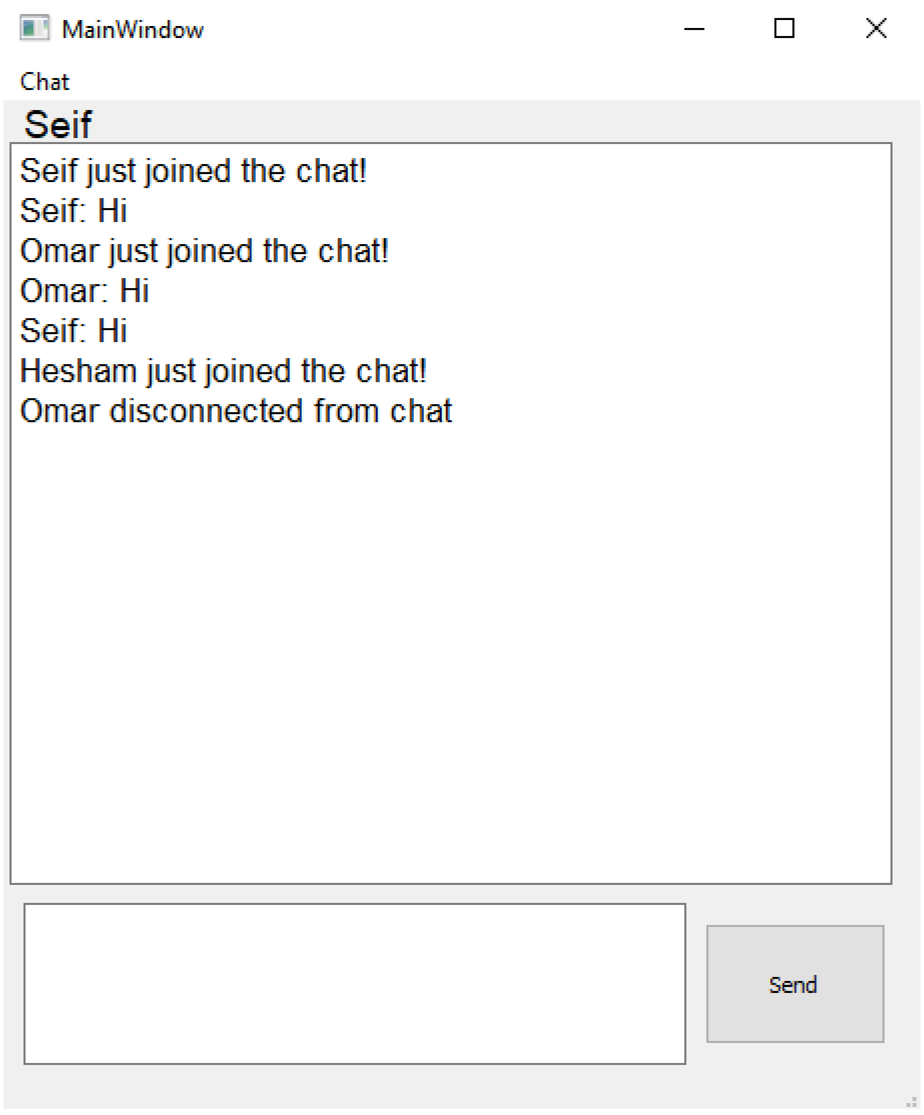
Hesham

Hesham just joined the chat!

Figure 12

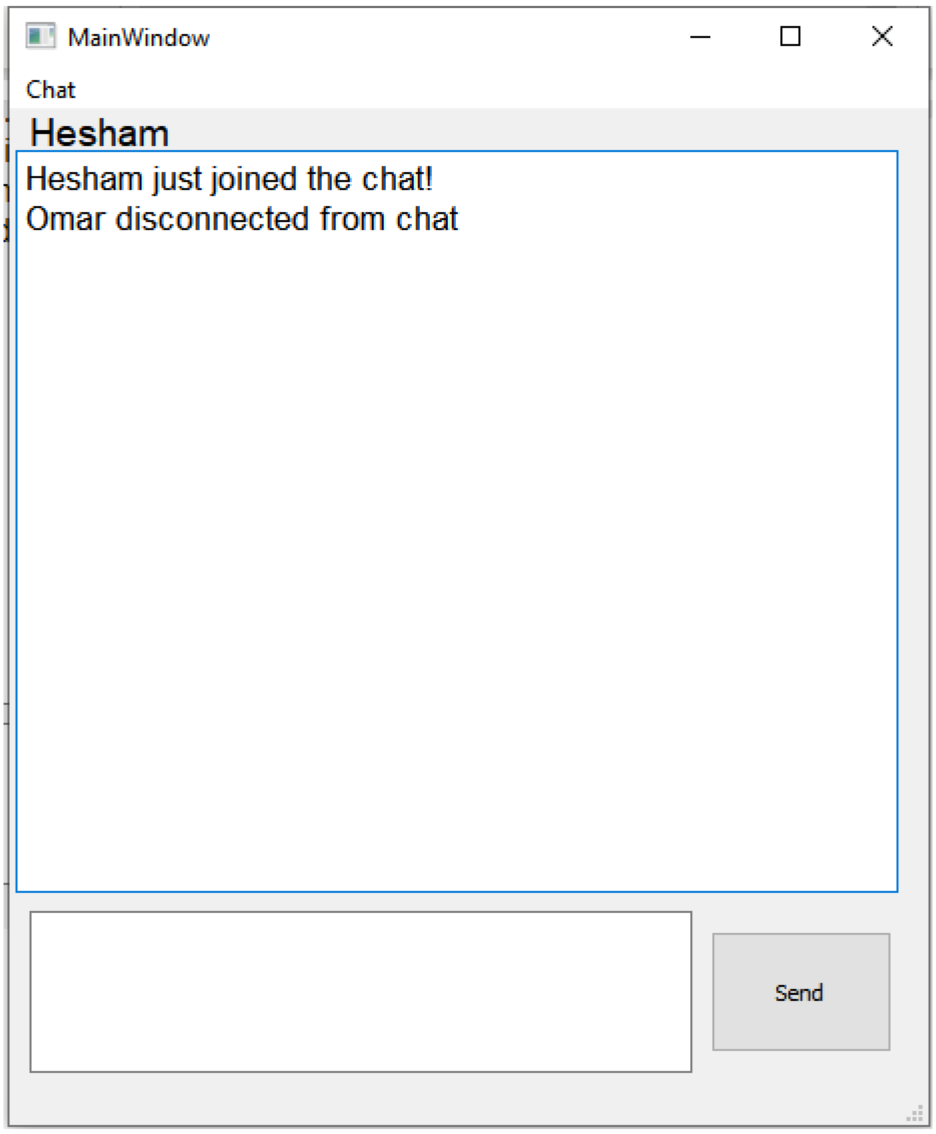
User is only shown messages sent after he enters the chat room.

When a user disconnects, all users in the chat room are notified.



```
Seif just joined the chat!
Seif: Hi
Omar just joined the chat!
Omar: Hi
Seif: Hi
Hesham just joined the chat!
Omar disconnected from chat
```

Figure 13



Hesham

Hesham just joined the chat!
Omar disconnected from chat

Figure 14

Controller

We have 2 types of controllers: 1- Car Controller, 2- Chat Controller.

The controller is responsible for sending and receiving data from the client to the server and displaying it in its respective GUI.

The car controller is responsible for getting data from car server and rendering data onto Client Car GUI.

The chat controller is responsible for getting data from chat server and rendering data onto Client Chat GUI.

Server Side

We have 2 multithreaded servers: 1- Chat Server, 2- Car Server.

Each server contains a dispatcher that creates worker threads that are responsible for serving multiple clients concurrently.

Chat Server sends any messages sent to it to the rest of users connected and is responsible for detecting disconnected clients and notifying the rest of users.

Car Server receives each player's information and sends it to the rest of the players in the room.

Servers are responsible for connecting with database and updating the database periodically as means of backup and robustness.

When all a room's players disconnect or when a room ends by declaring a winner, the room is deleted from database by server.

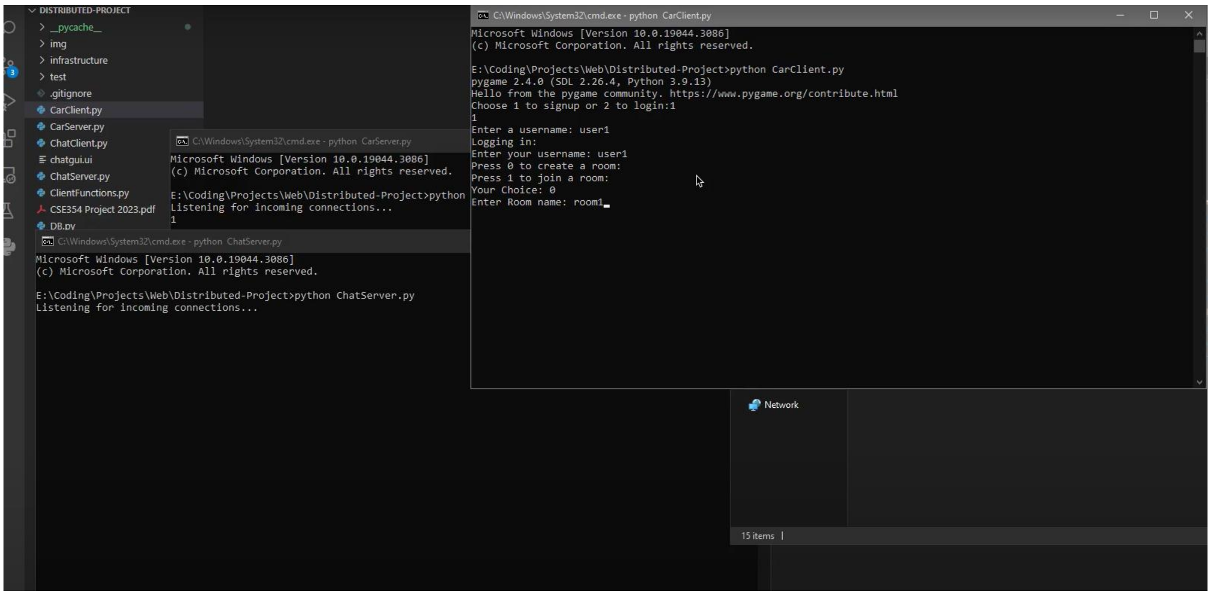
Clients can sign up or login by unique usernames through server. Server checks database for all usernames available and maintains a recent copy and regularly updates database when a new user signs up.

Each worker thread communicates with the database through a common database port.

All clients connect to car server through port 10000 and chat server through port 10001.

Testing scenarios and results

Running both servers and creating a player called user1 and a room that has 2 players.



C:\Windows\System32\cmd.exe - python CarClient.py

```
Microsoft Windows [Version 10.0.19044.3086]
(c) Microsoft Corporation. All rights reserved.

E:\Coding\Projects\Web\Distributed-Project>python CarClient.py
pygame 2.4.0 (SDL 2.26.4, Python 3.9.13)
Hello from the pygame community. https://www.pygame.org/contribute.html
Choose 1 to signup or 2 to login:1
1
Enter a username: user1
Logging in:
Enter your username: user1
Press 0 to create a room:
Press 1 to join a room:
Your Choice: 0
Enter Room name: room1
```

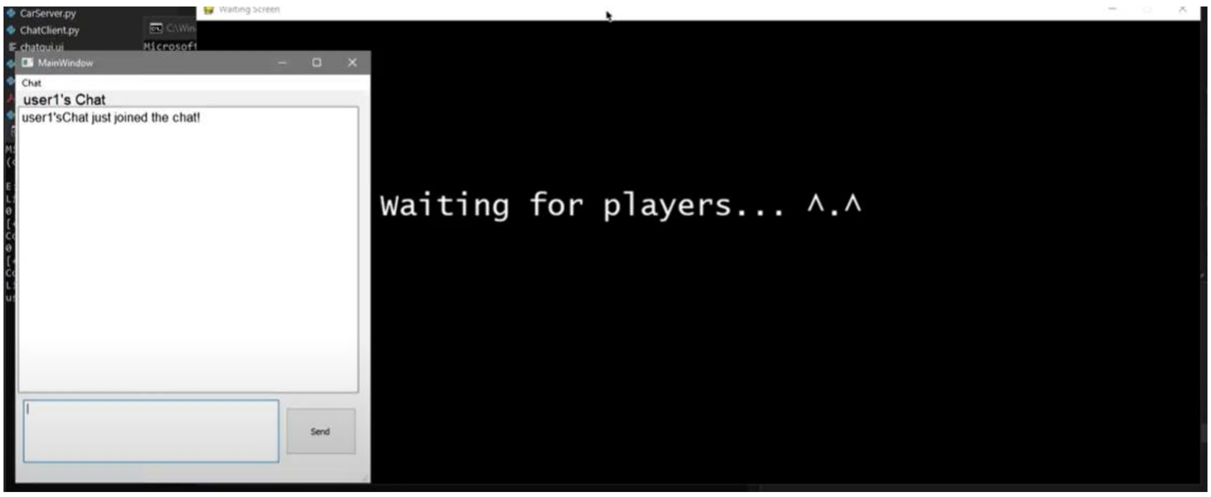
C:\Windows\System32\cmd.exe - python CarServer.py

```
Microsoft Windows [Version 10.0.19044.3086]
(c) Microsoft Corporation. All rights reserved.

E:\Coding\Projects\Web\Distributed-Project>python
Listening for incoming connections...
1
```

C:\Windows\System32\cmd.exe - python ChatServer.py

```
Microsoft Windows [Version 10.0.19044.3086]
(c) Microsoft Corporation. All rights reserved.

E:\Coding\Projects\Web\Distributed-Project>python ChatServer.py
Listening for incoming connections...
```

User1 waiting screen his chat shown.

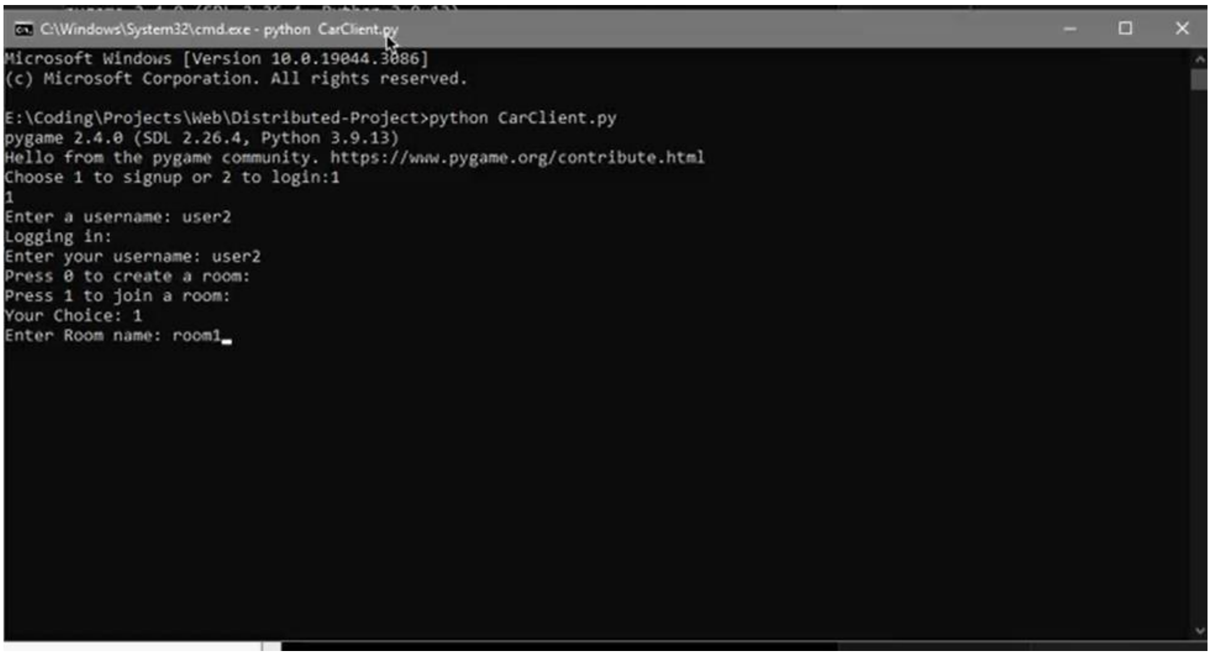


user1's Chat

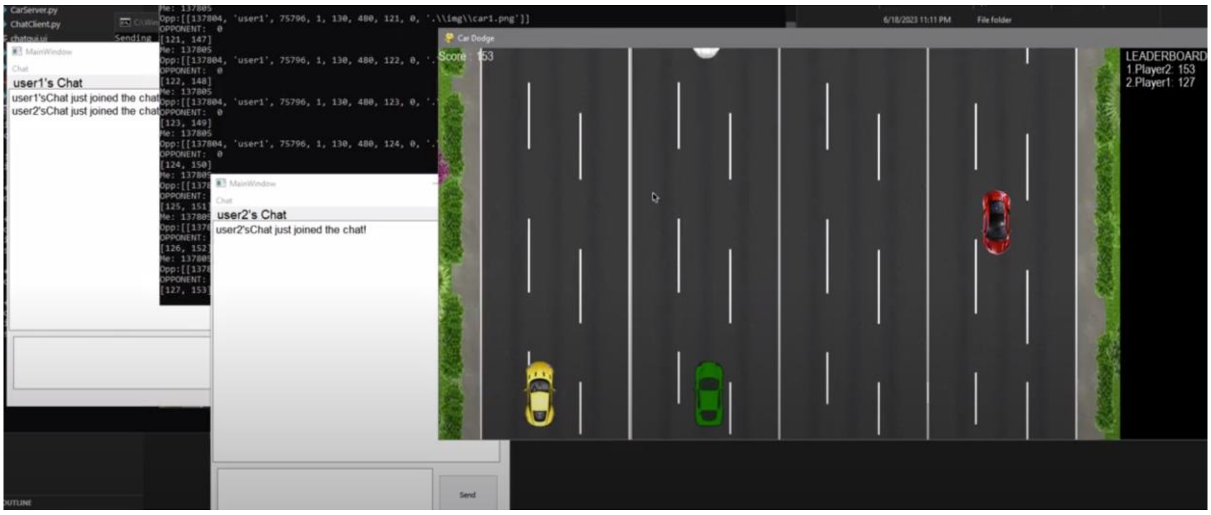
user1'sChat just joined the chat!

Waiting for players... ^.^

Player 2 joins



```
Microsoft Windows [Version 10.0.19044.3086]
(c) Microsoft Corporation. All rights reserved.

E:\Coding\Projects\Web\Distributed-Project>python CarClient.py
pygame 2.4.0 (SDL 2.26.4, Python 3.9.13)
Hello from the pygame community. https://www.pygame.org/contribute.html
Choose 1 to signup or 2 to login:1
1
Enter a username: user2
Logging in:
Enter your username: user2
Press 0 to create a room:
Press 1 to join a room:
Your Choice: 1
Enter Room name: room1
```



```
Me: 137805
Opp:[[137804, 'user1', 75796, 1, 130, 480, 121, 0, '.\\img\\car1.png']]
OPPONENT:  0
[121, 147]
Me: 137805
Opp:[[137804, 'user1', 75796, 1, 130, 480, 122, 0, '.
OPPONENT:  0
[122, 148]
Me: 137805
Opp:[[137804, 'user1', 75796, 1, 130, 480, 123, 0, '.
OPPONENT:  0
[123, 149]
Me: 137805
Opp:[[137804, 'user1', 75796, 1, 130, 480, 124, 0, '.
OPPONENT:  0
[124, 150]
```

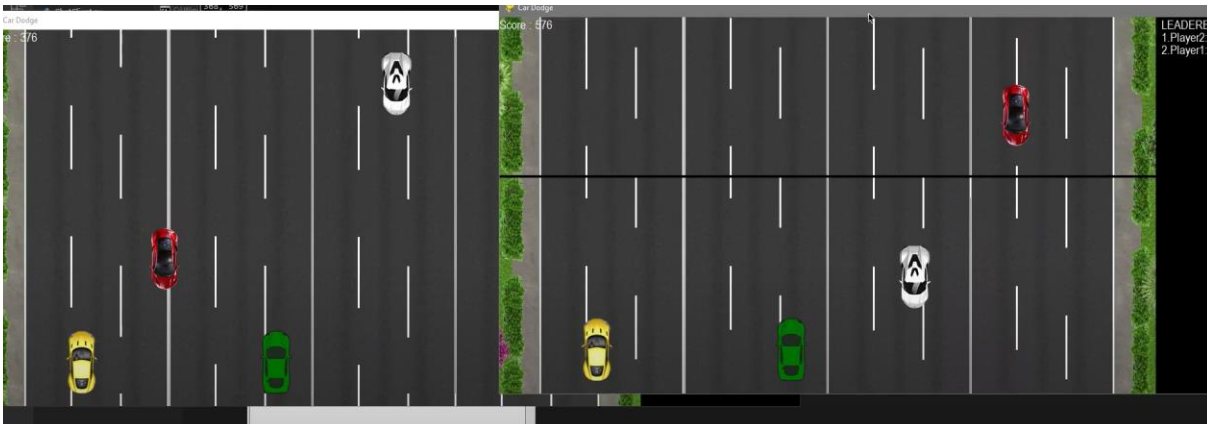
user1's Chat

user1'sChat just joined the chat
user2'sChat just joined the chat

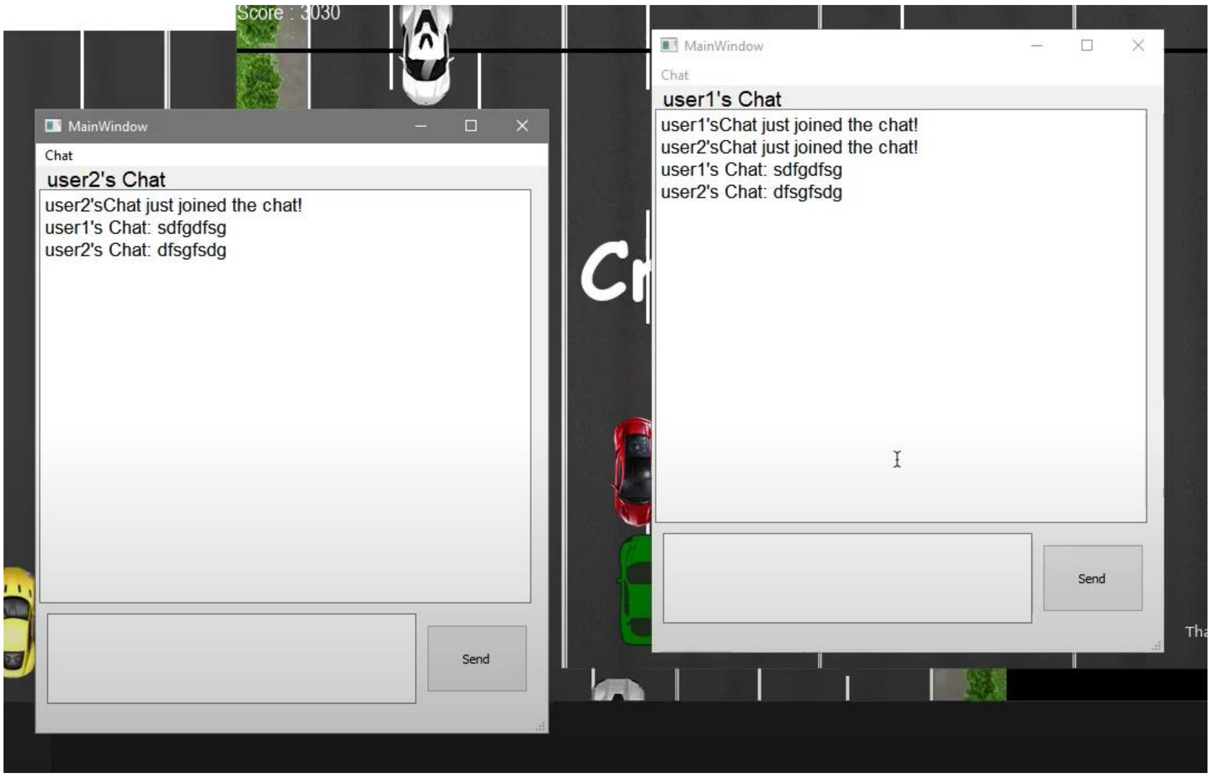
user2's Chat

user2'sChat just joined the chat!

Each Player can sees the other player mobbing and scores are displayed



Chatting feature typing anything will be shown to both also whomever joins first will see all the players that will join next by getting the message (player name) joined the chat!



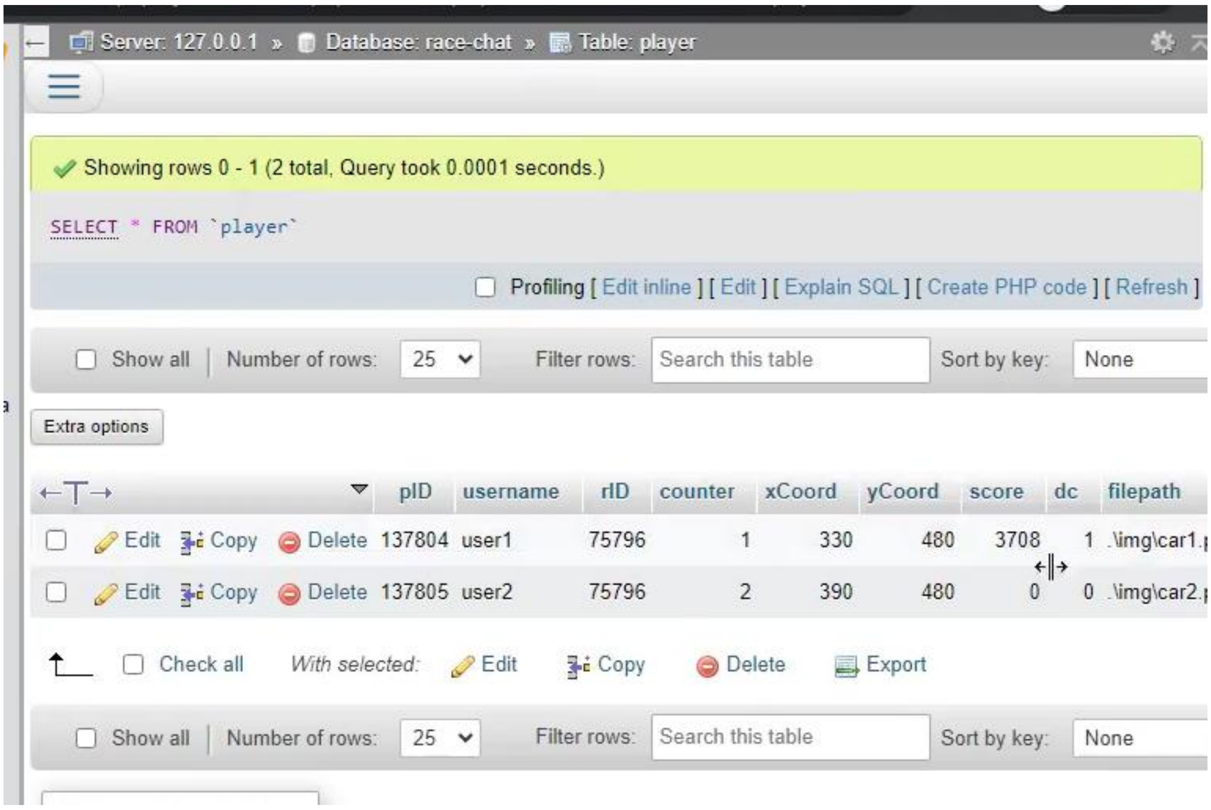
user2's Chat

```
user2'sChat just joined the chat!
user1's Chat: sdfgdfsg
user2's Chat: dfsgfsdg
```

user1's Chat

```
user1'sChat just joined the chat!
user2'sChat just joined the chat!
user1's Chat: sdfgdfsg
user2's Chat: dfsgfsdg
```

Player's info is saved in database in real time


Server: 127.0.0.1 » Database: race-chat » Table: player

✓ Showing rows 0 - 1 (2 total, Query took 0.0001 seconds.)

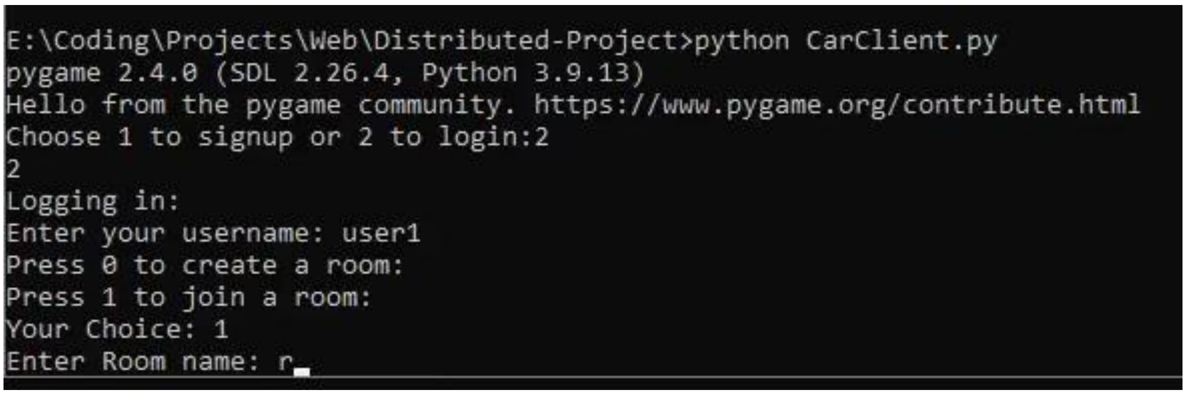
```
SELECT * FROM `player`
```

Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]
Show all | Number of rows: 25 Filter rows: Search this table Sort by key: None
Extra options

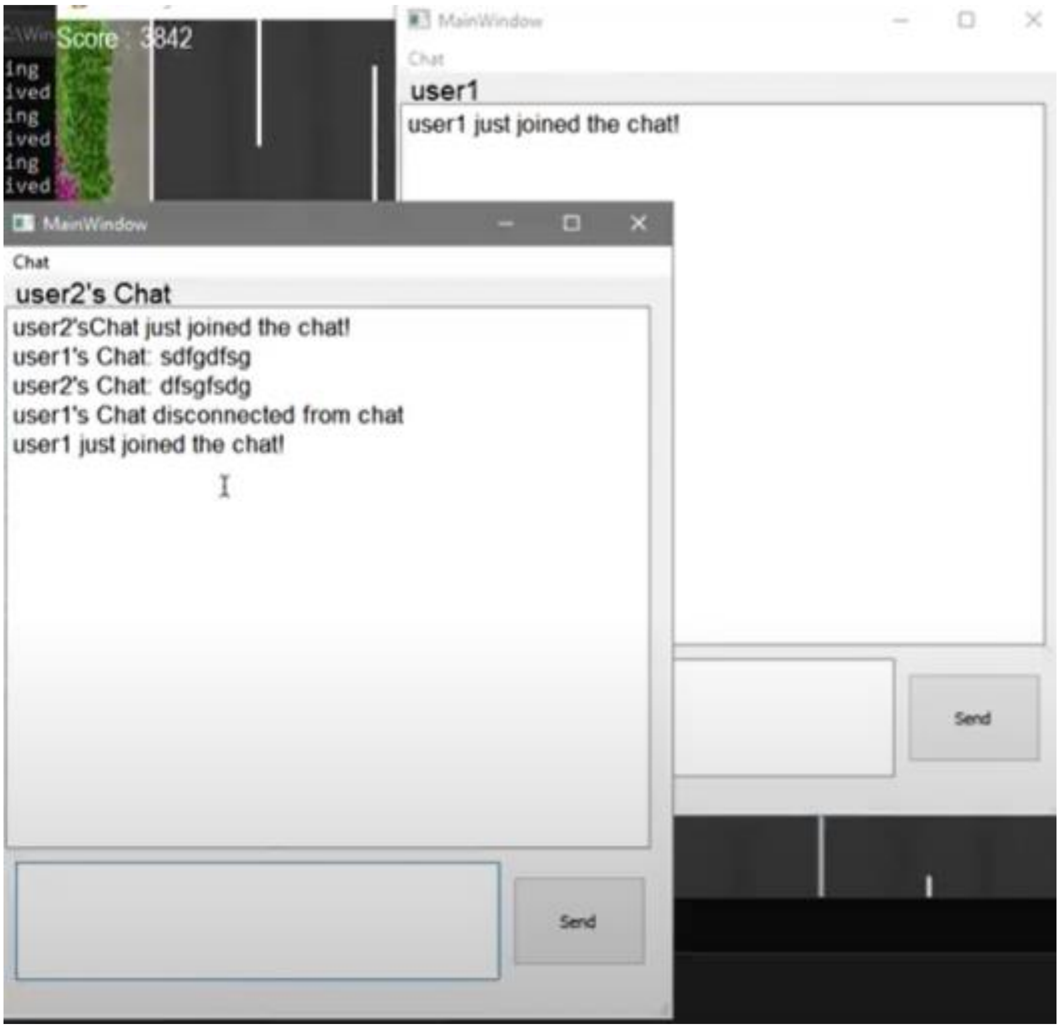
	pID	username	rID	counter	xCoord	yCoord	score	dc	filepath
Edit Copy Delete	137804	user1	75796	1	330	480	3708	1	.\img\car1.p
Edit Copy Delete	137805	user2	75796	2	390	480	0	0	.\img\car2.p

Check all With selected: Edit Copy Delete Export
Show all | Number of rows: 25 Filter rows: Search this table Sort by key: None

If user disconnects, they can log in again


```
E:\Coding\Projects\Web\Distributed-Project>python CarClient.py
pygame 2.4.0 (SDL 2.26.4, Python 3.9.13)
Hello from the pygame community. https://www.pygame.org/contribute.html
Choose 1 to signup or 2 to login:2
2
Logging in:
Enter your username: user1
Press 0 to create a room:
Press 1 to join a room:
Your Choice: 1
Enter Room name: r
```

He can rejoin the chat



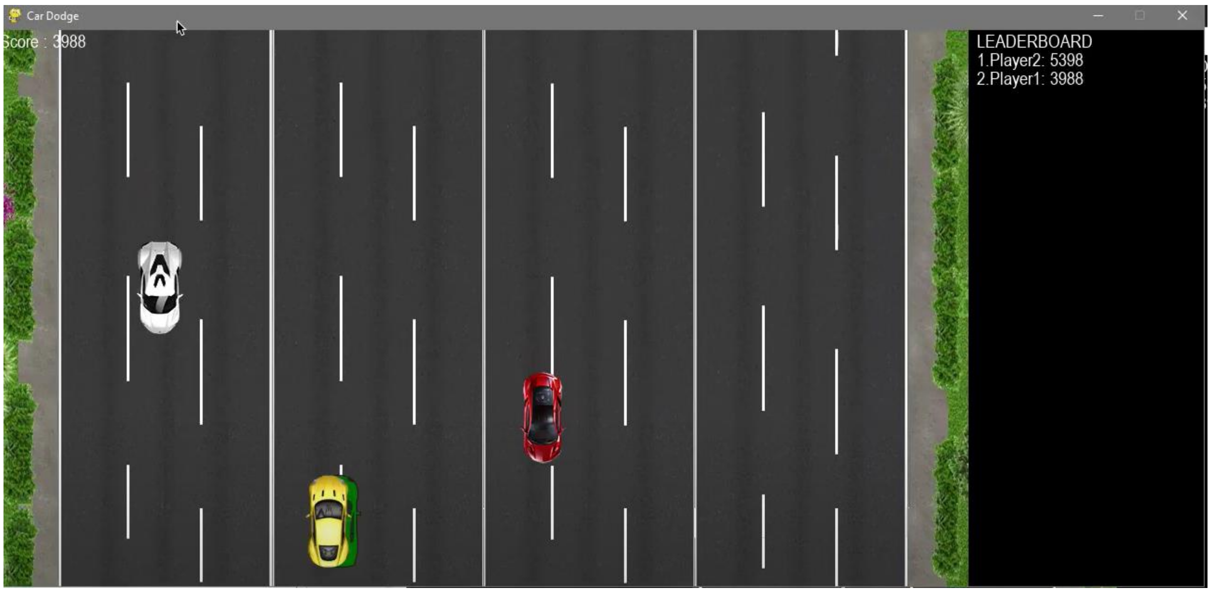
user1

user1 just joined the chat!

user2's Chat

```
user2'sChat just joined the chat!
user1's Chat: sdfgdfsg
user2's Chat: dfsgfsdg
user1's Chat disconnected from chat
user1 just joined the chat!
```

And can continue gaming with their last score before losing connection.



```
LEADERBOARD
1.Player2: 5398
2.Player1: 3988
```

End-user guide

1- Run the servers (carserver.py, chatserver.py)

2- Run CarClient.py

3- Follow cmd instructions

4- To add more players open new cmd and call python CarClient.py and follow instructions

libraries needed: pygame, socket, pickle and threading

Conclusion

In conclusion, the distributed car game with chat feature successfully leverages distributed computing concepts to provide an immersive and interactive gaming experience. By adopting a distributed architecture, the game achieves scalability, fault tolerance, and efficient resource utilization.

The distributed nature of the game allows for seamless gameplay across multiple machines or nodes in a network. Players can join the game from different locations, interact with each other, and compete in real-time. The game effectively utilizes load balancing techniques to evenly distribute the workload and ensure a smooth gaming experience for all participants.

Furthermore, the chat feature enhances the social aspect of the game by enabling players to communicate and strategize with each other. The distributed nature of the chat feature ensures that messages are delivered reliably and efficiently across the network, facilitating seamless communication among players regardless of their physical locations.

The fault tolerance capabilities of the distributed architecture ensure that the game remains resilient even in the face of node failures or network disruptions. The game can dynamically adapt to changes in the network, seamlessly redistributing game logic and resources to maintain uninterrupted gameplay.

Overall, the distributed car game with chat feature demonstrates the power and advantages of distributed computing in the gaming domain. It showcases the ability to create engaging and interactive experiences that transcend physical boundaries, bringing players together in a virtual world. Through efficient resource management, fault tolerance, scalability, and real-time communication, the distributed architecture of the game delivers an enjoyable and immersive gaming experience for players.

References:

1- Docker Swarm documentation: https://docs.docker.com/engine/swarm/

2- Docker Swarm Mode: Introduction to Swarm Mode in Docker: https://docs.docker.com/engine/swarm/swarm-mode/

3- Docker Swarm Services: https://docs.docker.com/engine/swarm/services/

4- Docker Swarm Load Balancing: https://docs.docker.com/engine/swarm/ingress/

5- Docker Swarm Scaling: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/

6- Docker Swarm Rolling Updates: https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/

7- Docker Swarm Secrets Management: https://docs.docker.com/engine/swarm/secrets/

8- Kubernetes Documentation: https://kubernetes.io/docs/

Kubernetes Concepts: https://kubernetes.io/docs/concepts/

9- Kubernetes Architecture: https://kubernetes.io/docs/concepts/architecture/

10 -Kubernetes Pods: https://kubernetes.io/docs/concepts/workloads/pods/

11 -Kubernetes Services: https://kubernetes.io/docs/concepts/services-networking/service/

12 -Kubernetes Scaling: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

13 -Kubernetes Configuration and Secrets: https://kubernetes.io/docs/concepts/configuration/

14 -Kubernetes Rolling Updates and Rollbacks: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

15 -ANDREW S. TANENBAUM, MAARTEN VAN STEEN - Distributed Systems, Principles And Paradigms,Prentice Hall