Distributed Systems Lab
Car Hire Booking Distributed System

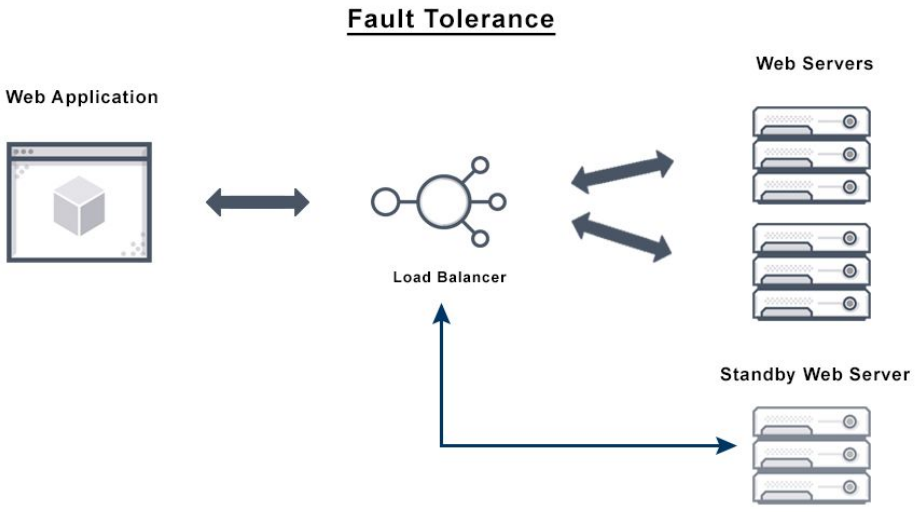
Team Members:
Tania Rajabally (2017130047)
Shruti Rampure(2017130048)
Rahul Ramteke(2017130049)

Batch: C

Problem Statement:
The purpose of this system is to monitor and control the bookings of cars in a distributed environment. All the features of a typical Car Hire Booking System are discussed here by considering a distributed system.

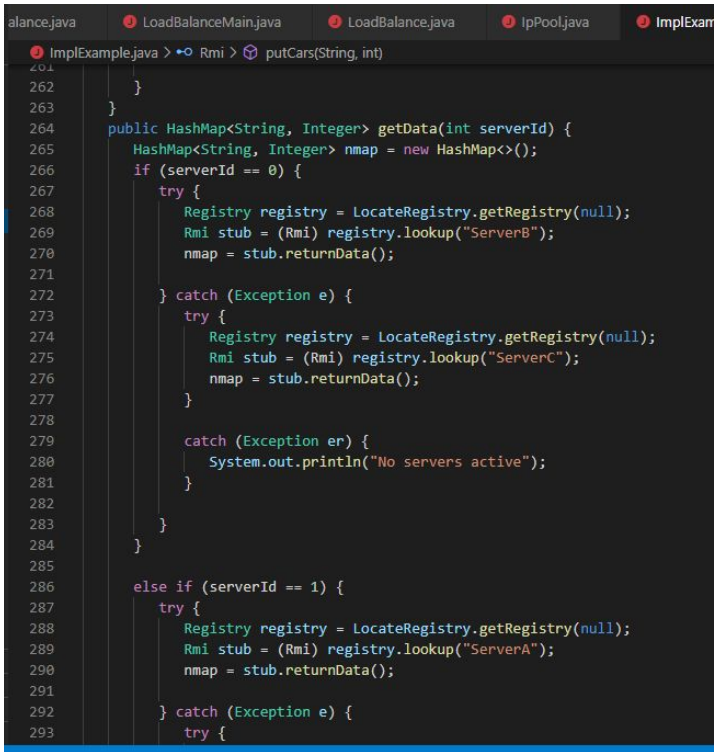
FAULT TOLERANCE

Fault tolerance refers to the ability of a system (computer, network, cloud cluster, etc.) to continue operating without interruption when one or more of its components fail.
The objective of creating a fault-tolerant system is to prevent disruptions arising from a single point of failure, ensuring the high availability and business continuity of mission-critical applications or systems.
Fault-tolerant systems use backup components that automatically take the place of failed components, ensuring no loss of service.
A fault-tolerant design enables a system to continue its intended operation, possibly at a reduced level, rather than failing completely, when some part of the system fails.[2] The term is most commonly used to describe computer systems designed to continue more or less fully operational with, perhaps, a reduction in throughput or an increase in response time in the event of some partial failure. That is, the system as a whole is not stopped due to problems either in the hardware or the software.



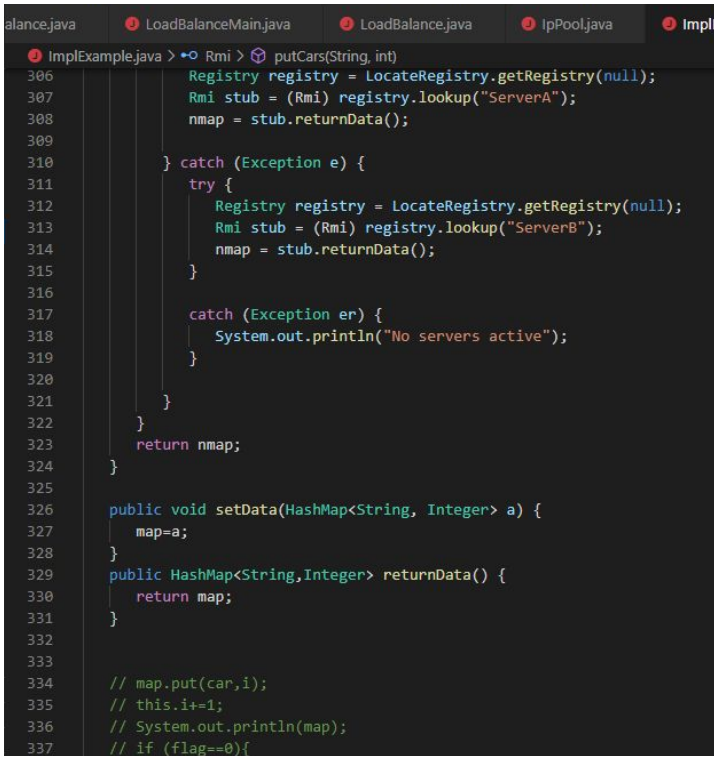
Code:

Implementation.java

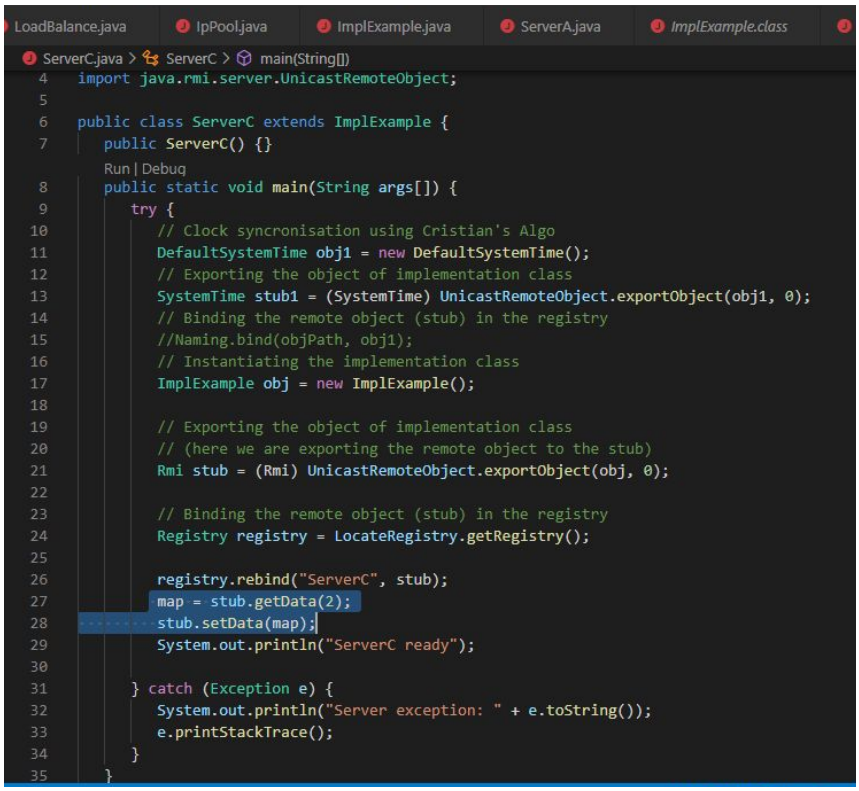


```
    }
}
public HashMap<String, Integer> getData(int serverId) {
    HashMap<String, Integer> nmap = new HashMap<>();
    if (serverId == 0) {
        try {
            Registry registry = LocateRegistry.getRegistry(null);
            Rmi stub = (Rmi) registry.lookup("ServerB");
            nmap = stub.returnData();

        } catch (Exception e) {
            try {
                Registry registry = LocateRegistry.getRegistry(null);
                Rmi stub = (Rmi) registry.lookup("ServerC");
                nmap = stub.returnData();
            }

            catch (Exception er) {
                System.out.println("No servers active");
            }

        }
    }

    else if (serverId == 1) {
        try {
            Registry registry = LocateRegistry.getRegistry(null);
            Rmi stub = (Rmi) registry.lookup("ServerA");
            nmap = stub.returnData();

        } catch (Exception e) {
            try {
```

```
            Registry registry = LocateRegistry.getRegistry(null);
            Rmi stub = (Rmi) registry.lookup("ServerA");
            nmap = stub.returnData();

        } catch (Exception e) {
            try {
                Registry registry = LocateRegistry.getRegistry(null);
                Rmi stub = (Rmi) registry.lookup("ServerC");
                nmap = stub.returnData();
            }

            catch (Exception er) {
                System.out.println("No servers active");
            }

        }
    } else if (serverId == 2) {
        try {
            Registry registry = LocateRegistry.getRegistry(null);
            Rmi stub = (Rmi) registry.lookup("ServerA");
            nmap = stub.returnData();

        } catch (Exception e) {
            try {
                Registry registry = LocateRegistry.getRegistry(null);
                Rmi stub = (Rmi) registry.lookup("ServerB");
                nmap = stub.returnData();
            }

            catch (Exception er) {
                System.out.println("No servers active");
            }
```



ServerC.java



Explanation of Implementation:

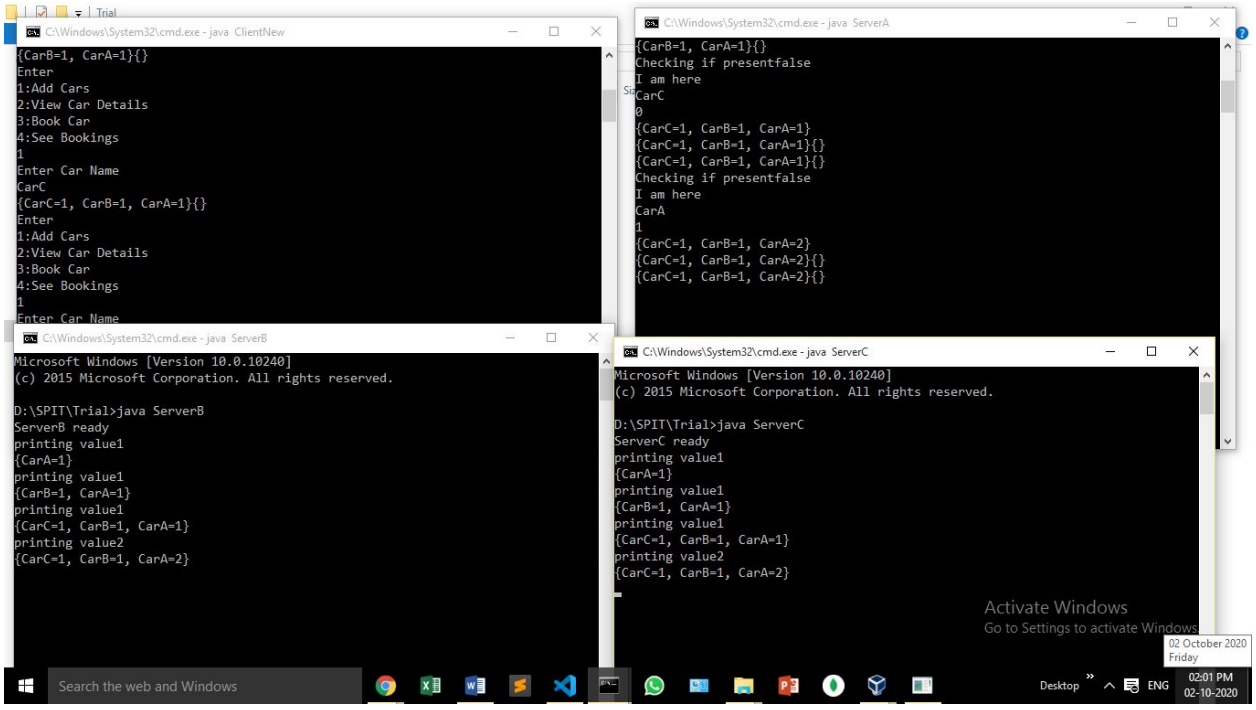
Each time a change or a new request is made to the server, it first checks if data already exists in the other server. If it does, it retrieves this data from the other server. If no other server is present or the data is the same, it adds the new data according to the new client request made. If one server crashes, then when it starts working again, it first checks the data store in the other replication servers and retrieves that data. In this way all the replication servers have the same data, no data is lost and data consistency is maintained.

When a server crashes, we lookup for the other servers, use the getData() function to get the data from the other replication servers and then use the setData() function to set the data in the server that has crashed. Every new request now will have the same data stored in all three replication servers and it will work as previously before one of the servers crashed.

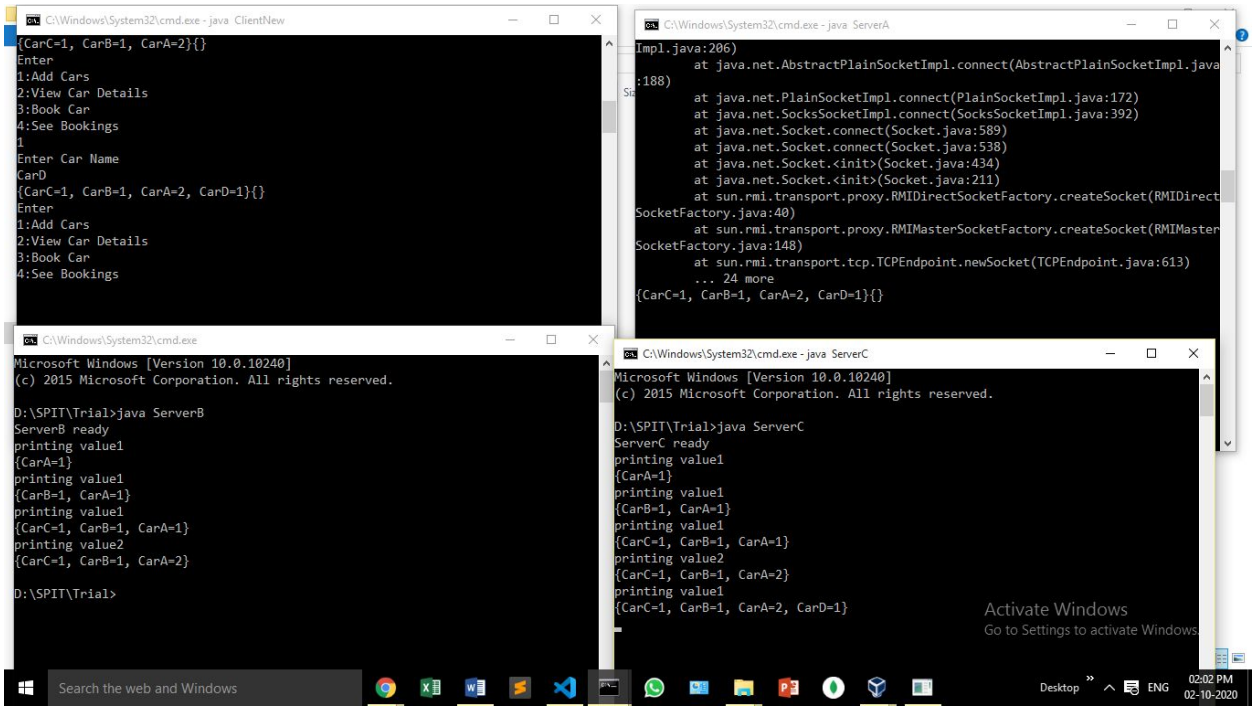
Steps to run:

1. Start rmiregistry
2. Compile the files : javac *.java
3. Run the command:
   java ServerA so that the serverA will be ready.
   In a new window run :
   Java ServerB so that serverB is ready.
   In a new window run :
   Java ServerC so that serverC is ready.
4. In a new window, run the command: java ClientNew to run the client side program.
5. After adding a few cars, shut one server.
6. Now restart that server and see that the data will be retrieved.

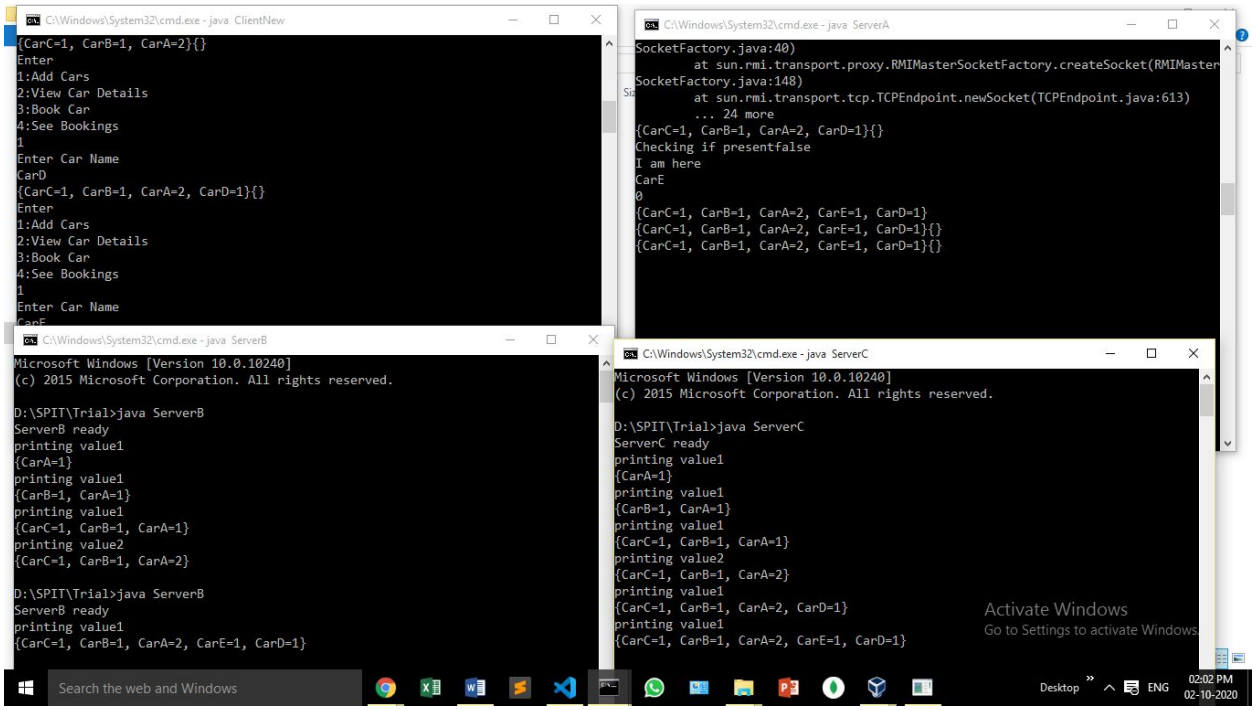
Screenshots:



This is when all the servers are working normally and we have added a few cars which have been added to all the replication servers.



Now if serverB crashes, the data will be added to the other two servers. ServerA informs that replication serverB is not connected but it still updates its data stored according to the new request.



Now we restart serverB. We can see that it retrieved the data from the other replication servers and now data consistency is maintained. Now every new request will work normally like before the server crashed.

Conclusion:

We implemented fault tolerance and saw how it is handled when a particular replication server crashes, how data consistency is maintained and how data loss is handled. Thus data will always be consistent in our system. In this manner, fault tolerance was implemented and integrated in our car booking system.