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.

DATA CONSISTENCY

Consistency in database systems refers to the requirement that any given database transaction must change affected data only in allowed ways. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined database constraints.
Data Consistency refers to the usability of data. Data Consistency problems may arise even in a single-site environment during recovery situations when backup copies of the production data are used in place of the original data.
In order to ensure that your backup data is usable, it is necessary to understand the backup methodologies that are in place as well as how the primary data is created and accessed. Another very important consideration is the consistency of the data once the recovery has been completed and the application is ready to begin processing.

Code:

ClientNew.java

```java
int choice =0;
   while(choice!=-1){

      String answer;
      System.out.println("Enter "+"\n"+"1:Add Cars"+"\n"+"2:View Car Details"+"\n"+"3:Book Car"+"\n"+"4:See E
      choice = myObj.nextInt();
      if (choice==1){
        System.out.println("Enter Car Name");
        // Calling the remote method using the obtained object
        String carName = myObj.next();
        stub.putCars(carName,0);
        System.out.println(stub.printMsg());
     }
     else if(choice ==2){
        System.out.println("The Cars available are:");
        System.out.println(stub.printMsg());
     }
     else if (choice==3){
      System.out.println("Enter the User's name");
      String userName = myObj.next();
        System.out.println("Enter the car name to book");
        String carName = myObj.next();
        System.out.println("Enter the date + time");
        String time = myObj.next();
        stub.bookCar(userName,carName,time,0);
        // System.out.println(answer);
     }
     else if (choice==4){
```

Implementation.java

```java
   void bookCar(String userName, String carName, String time, int flag) throws RemoteException;
}

public class ImplExample implements Rmi {
   private static Integer position = 1;
   private static Integer limit = 0;
   static HashMap<String, Integer> map = new HashMap<>();
   static HashMap<String, ArrayList> bookingmap = new HashMap<>();
   static int i = 0;

   // Implementing the interface method
   public String printMsg() {
      return map.toString() + bookingmap.toString();
   }

   public void replicatePut(String car, int value) {
      System.out.println("printing value" + value);
      map.put(car, value);
      this.i += 1;
      System.out.println(map);
   }

   public void bookingPut(String car, ArrayList list) {
      // System.out.println("printing value"+value);
      bookingmap.put(car, list);
      this.i += 1;
      System.out.println(bookingmap);
   }

   public void putCars(String car, int flag) {
      Iterator<Map.Entry<String, Integer>> iterator = map.entrySet().iterator();

```

```java
        System.out.println(bookingmap);
    }

    public void putCars(String car, int flag) {
        Iterator<Map.Entry<String, Integer>> iterator = map.entrySet().iterator();

        // flag to store result
        boolean isKeyPresent = false;

        // Iterate over the HashMap
        while (iterator.hasNext()) {

            // Get the entry at this iteration
            Map.Entry<String, Integer> entry = iterator.next();

            // Check if this key is the required key
            if (car == entry.getKey()) {

                isKeyPresent = true;
            }
        }
        System.out.println("Checking if present" + isKeyPresent);

        int count = map.containsKey(car) ? map.get(car) : 0;
        map.put(car, count + 1);
        // map.put(car,i);

        // else{
        // map.put(car,1);
        // }
        // map.put(car,i);
        System.out.println("I am here");
        System.out.println(car);
```

ServerA.java

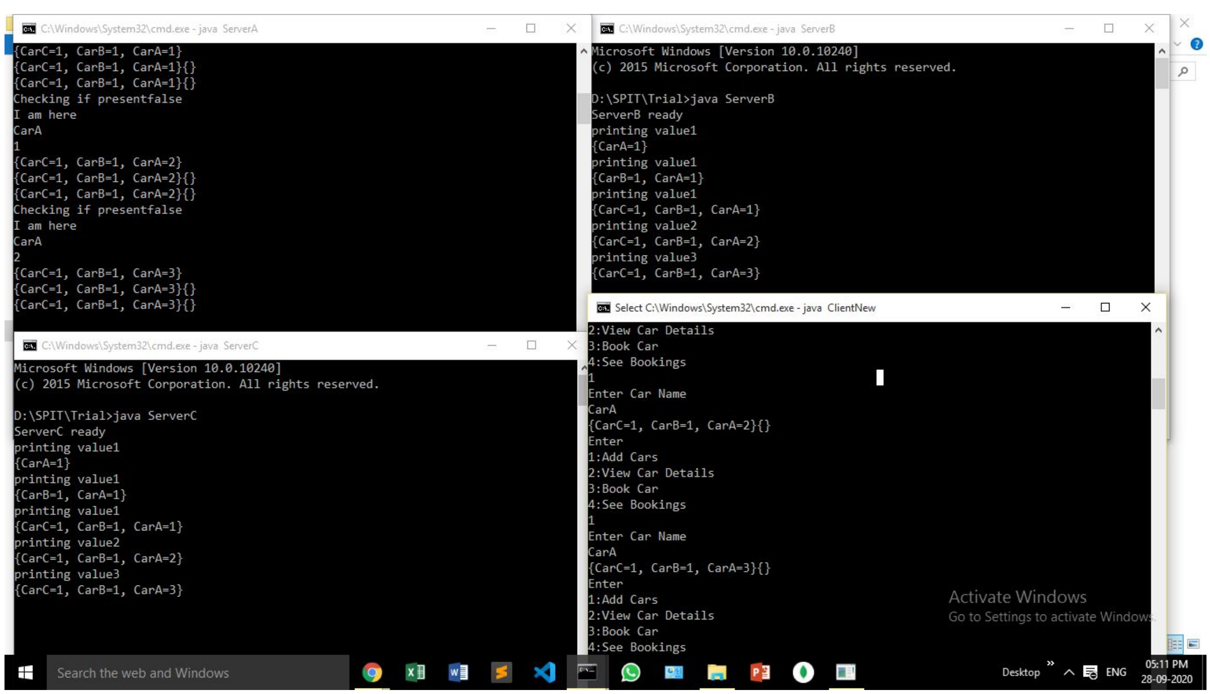
```java
import java.rmi.registry.*;

public class ServerA extends ImplExample {
    public ServerA() {}

    public static void main(String args[]) {
        String objPath = "//localhost:1099/SystemTime";
        try {
            // Instantiating the implementation class
            ImplExample obj = new ImplExample();

            DefaultSystemTime obj1 = new DefaultSystemTime();
            // Exporting the object of implementation class
            SystemTime stub1 = (SystemTime) UnicastRemoteObject.exportObject(obj1, 0);
            // Binding the remote object (stub) in the registry
            Naming.bind(objPath, obj1);


            // Exporting the object of implementation class
            // (here we are exporting the remote object to the stub)
            Rmi stub = (Rmi) UnicastRemoteObject.exportObject(obj, 0);

            // Binding the remote object (stub) in the registry
            Registry registry = LocateRegistry.getRegistry();

            registry.rebind("ServerA", stub);
            map = stub.getData(0);
            stub.setData(map);
            System.out.println("ServerA ready");

            Naming.rebind("ServerA",obj);
            LocateRegistry.createRegistry(1901);
            Naming.rebind("rmi://localhost:1901"+"/pikachu",obj);
```

Explanation of Implementation:
ServerA is the main server. Everytime a change is made to serverA, the same change is made to the other two servers that are serverB and serverC. In this way it is checked that data consistency is maintained. As observed in the screenshots below, any change made is replicated in all the servers and in this manner data consistency is maintained and all the three servers have the same data stored. When different clients request data at the same time, load balancing is implemented further for that which will be explained later.
Each time a change is made to the main server i.e serverA, like if a car is added or booking is done then that car has to be removed from the available cars, all these changes are reflected in all three servers and in this way consistency is maintained.
We can add new cars, view the cars stored in the database. When we make a booking, the user has to fill in their details and choose the car they would like to book. The availability of that car then reduces by 1. This happens in all three servers and hence data consistency is maintained.

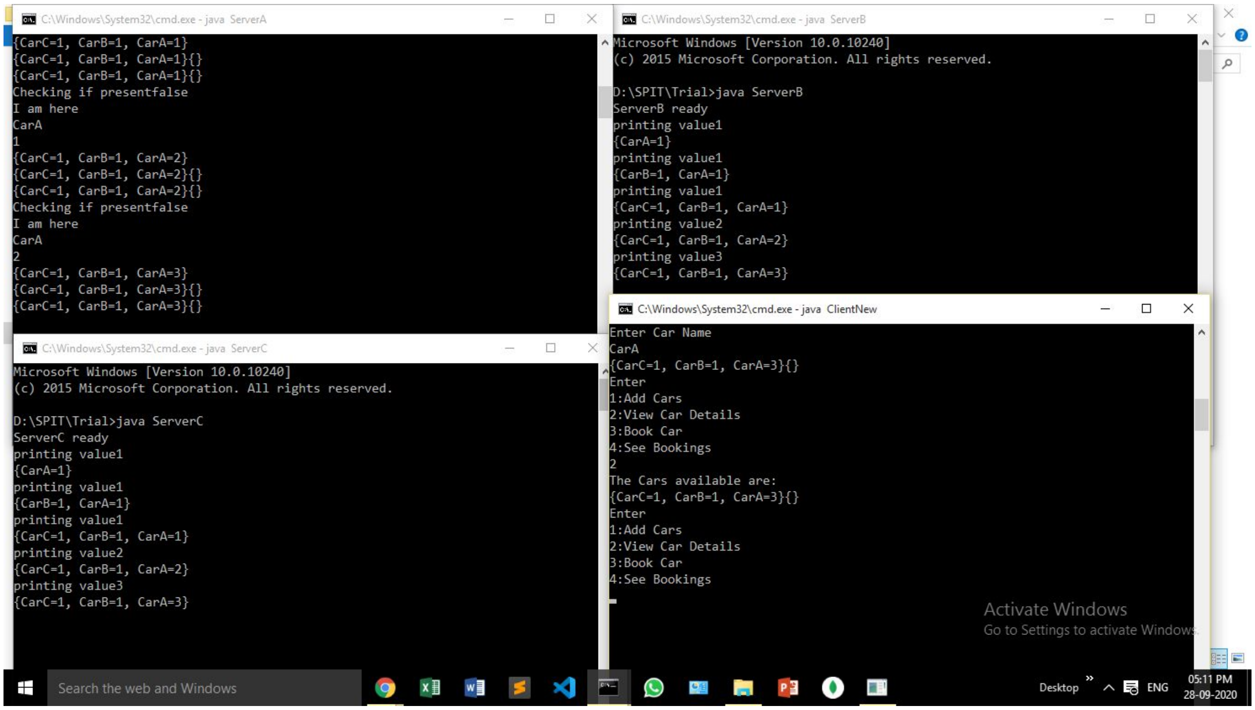
Steps to run:

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. The client can then add new cars.

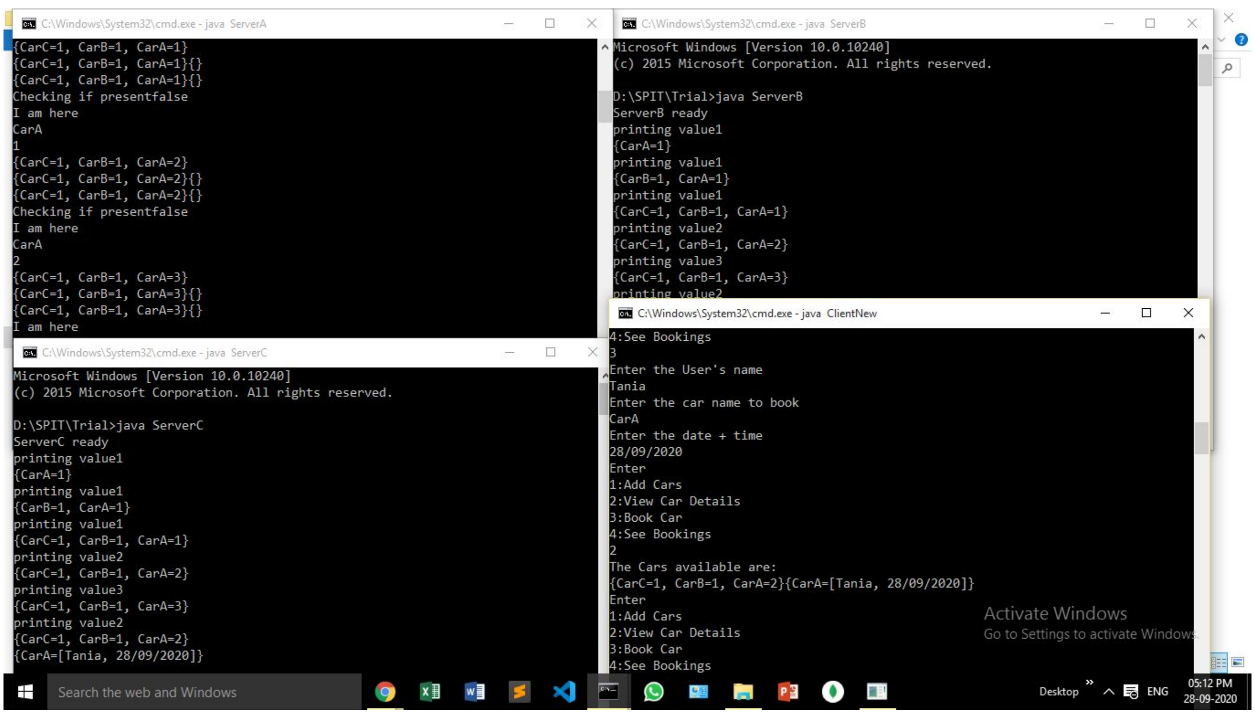
Screenshots:



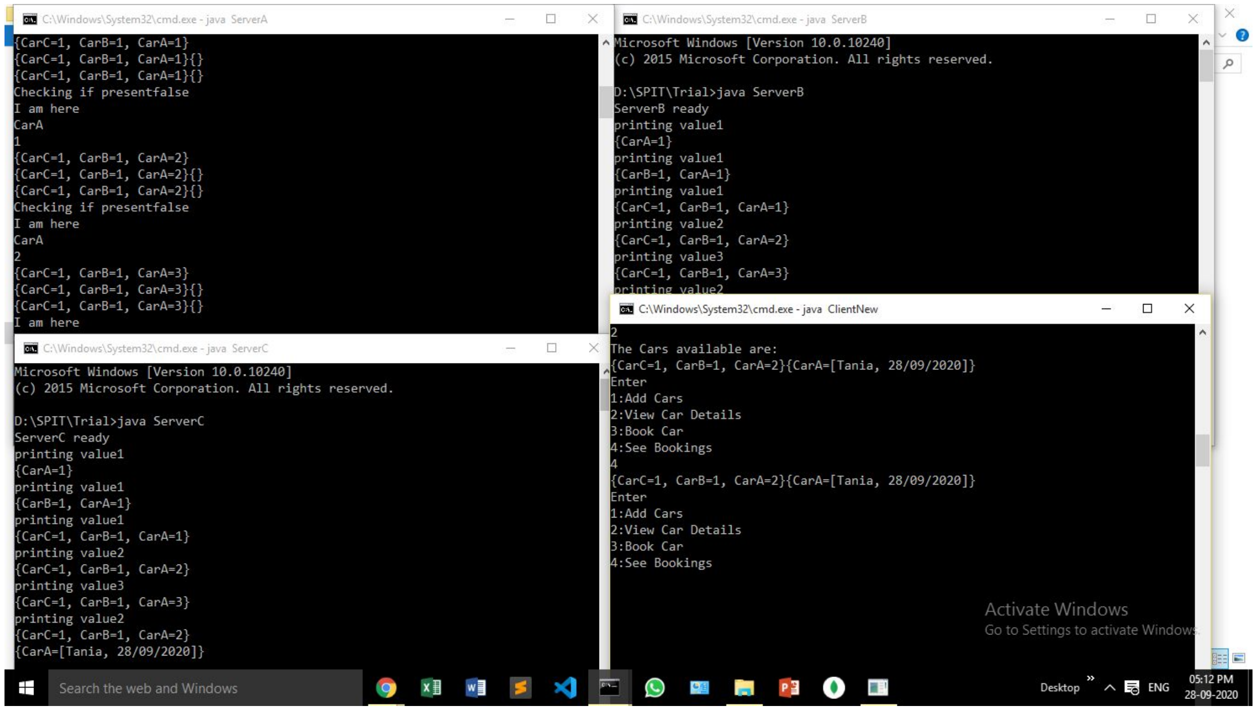
The new cars have been added to all the three servers and the data is consistent.



We can view the details of the cars that is stored in the hashmap that we created.



We can book a car for a particular day by filling in the details and choosing the car the user would like to book. This booking is stored in all the three servers and the data is consistent.



We can see that since a booking for carA was done, the availability of carA has reduced by 1 and now only 2 cars of carA are available. It can be noticed that this change appeared in all three servers i.e serverA, serverB and serverC. Hence data consistency was maintained.

Conclusion:

Data consistency is important as if multiple requests are made at the same time, the data stored should be consistent and not lead to conflicts and loss of data later. Hence, all the servers should have the same data before a transaction is carried out. Our car booking system has all the modules integrated and data consistency is maintained throughout the system.