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.

MUTUAL EXCLUSION

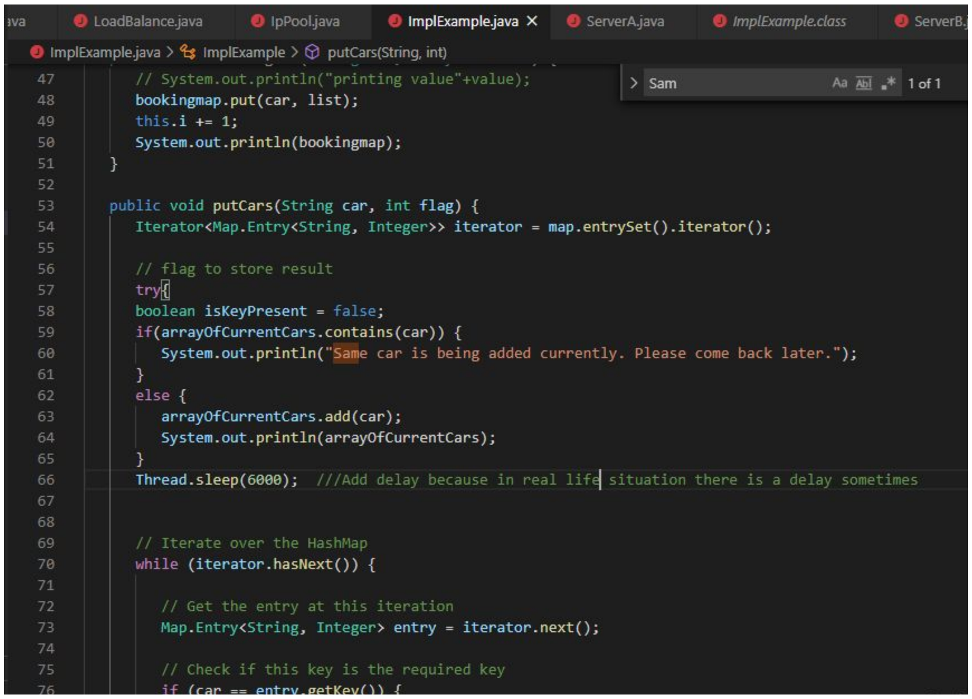
Mutual exclusion is a concurrency control property which is introduced to prevent race conditions. It is the requirement that a process can not enter its critical section while another concurrent process is currently present or executing in its critical section i.e only one process is allowed to execute the critical section at any given instance of time.
In Distributed systems, we neither have shared memory nor a common physical clock and there for we can not solve mutual exclusion problem using shared variables. To eliminate the mutual exclusion problem in distributed system approach based on message passing is used.
A site in distributed system do not have complete information of state of the system due to lack of shared memory and a common physical clock.

Requirements of Mutual exclusion Algorithm:

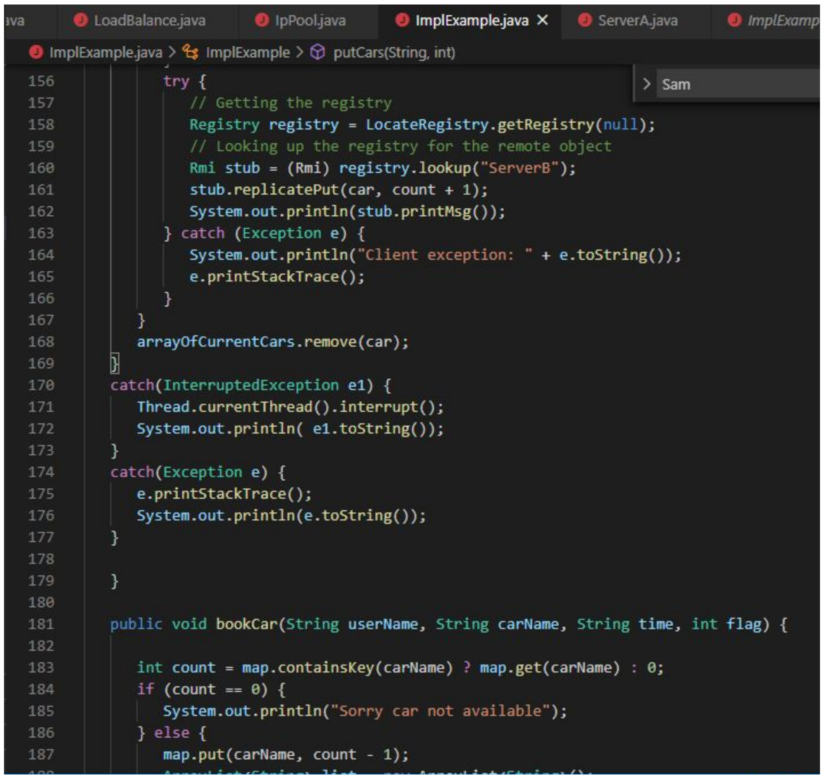
- No Deadlock:
  Two or more site should not endlessly wait for any message that will never arrive.
- No Starvation:
  Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section
- Fairness:
  Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.
- Fault Tolerance:
  In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

Code:

Implementation.java



```java
        // 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();

        // flag to store result
        try{
        boolean isKeyPresent = false;
        if(arrayOfCurrentCars.contains(car)) {
            System.out.println("Same car is being added currently. Please come back later.");
        }
        else {
            arrayOfCurrentCars.add(car);
            System.out.println(arrayOfCurrentCars);
        }
        Thread.sleep(6000);  ///Add delay because in real life situation there is a delay sometimes


        // 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()) {
```



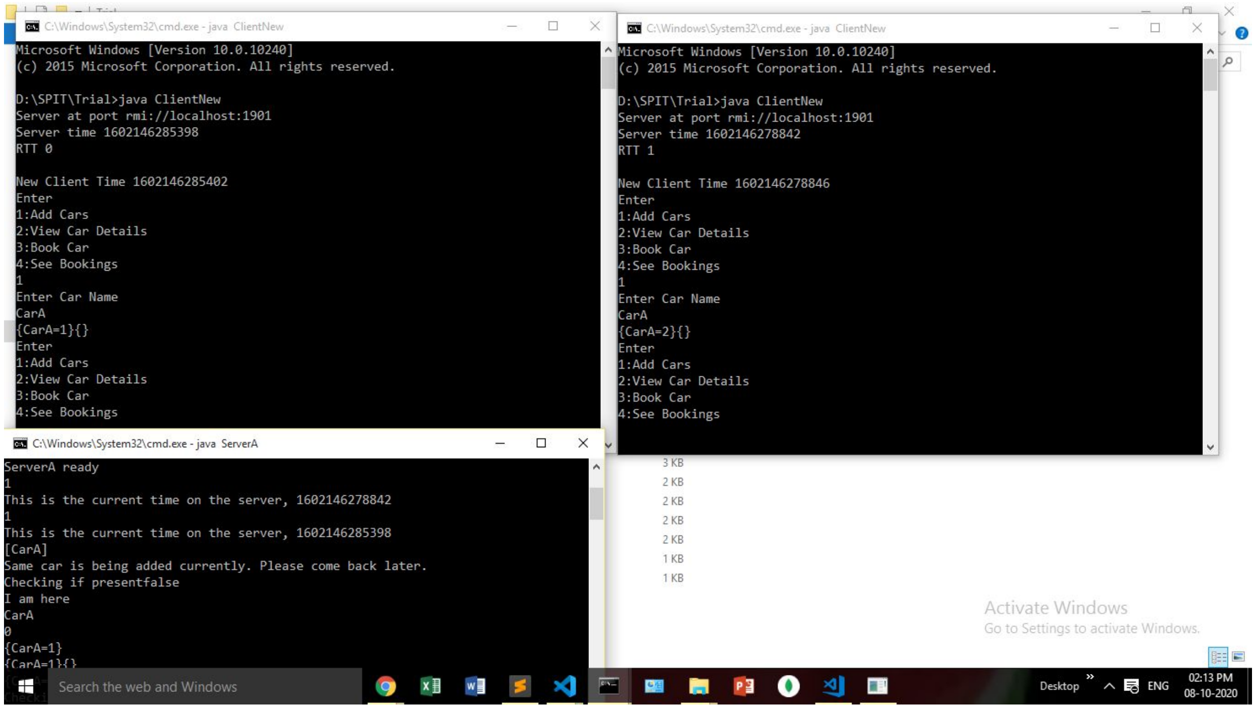
```java
            try {
                // Getting the registry
                Registry registry = LocateRegistry.getRegistry(null);
                // Looking up the registry for the remote object
                Rmi stub = (Rmi) registry.lookup("ServerB");
                stub.replicatePut(car, count + 1);
                System.out.println(stub.printMsg());
            } catch (Exception e) {
                System.out.println("Client exception: " + e.toString());
                e.printStackTrace();
            }
        }
        arrayOfCurrentCars.remove(car);
    }
    catch(InterruptedException e1) {
        Thread.currentThread().interrupt();
        System.out.println( e1.toString());
    }
    catch(Exception e) {
        e.printStackTrace();
        System.out.println(e.toString());
    }

    }

    public void bookCar(String userName, String carName, String time, int flag) {

        int count = map.containsKey(carName) ? map.get(carName) : 0;
        if (count == 0) {
            System.out.println("Sorry car not available");
        } else {
            map.put(carName, count - 1);
```

Explanation of Implementation:
When a car is being added, we purposely induce some delay by doing Thread.sleep(). Now if another client tries to add the same car, they will have to wait till the sleep time is not over and only then the car will be added. This will make sure that the data is consistent and no data is lost. In this way at once, only one client has access to that shared memory to add cars. This is how mutual exclusion works in our system.

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. Repeat this in another window.
6. Now add a car from one client. Now the thread will sleep
7. Now when you add the same car from the other client, it will wait till the first client completes adding and then the second request will be processed.

Screenshots:



```
C:\Windows\System32\cmd.exe - java  ClientNew
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

D:\SPIT\Trial>java ClientNew
Server at port rmi://localhost:1901
Server time 1602146285398
RTT 0

New Client Time 1602146285402
Enter
1:Add Cars
2:View Car Details
3:Book Car
4:See Bookings
1
Enter Car Name
CarA
{CarA=1}{}
Enter
1:Add Cars
2:View Car Details
3:Book Car
4:See Bookings
```

```
C:\Windows\System32\cmd.exe - java  ClientNew
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

D:\SPIT\Trial>java ClientNew
Server at port rmi://localhost:1901
Server time 1602146278842
RTT 1

New Client Time 1602146278846
Enter
1:Add Cars
2:View Car Details
3:Book Car
4:See Bookings
1
Enter Car Name
CarA
{CarA=2}{}
Enter
1:Add Cars
2:View Car Details
3:Book Car
4:See Bookings
```

```
C:\Windows\System32\cmd.exe - java  ServerA
ServerA ready
1
This is the current time on the server, 1602146278842
1
This is the current time on the server, 1602146285398
[CarA]
Same car is being added currently. Please come back later.
Checking if presentfalse
I am here
CarA
0
{CarA=1}
{CarA=1}{}
```

While the first client was adding CarA, the second client also tried adding CarA. The message was printed that the car is being added currently. After the first clients request was processed, the second clients request was processed and finally 2 CarA were added.

Conclusion:
We have implemented mutual exclusion so that data is not lost and two clients can't add the same cars at the same time. In this manner, mutual exclusion was implemented and integrated in our car booking system.