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.

CLOCK SYNCHRONIZATION

Distributed System is a collection of computers connected via the high speed communication network. In the distributed system, the hardware and software components communicate and coordinate their actions by message passing. Each node in distributed systems can share their resources with other nodes. So, there is a need for proper allocation of resources to preserve the state of resources and help coordinate between the several processes. To resolve such conflicts, synchronization is used. Synchronization in distributed systems is achieved via clocks. The physical clocks are used to adjust the time of nodes.Each node in the system can share its local time with other nodes in the system. The time is set based on UTC (Universal Time Coordination). UTC is used as a reference time clock for the nodes in the system.
The clock synchronization can be achieved by 2 ways: External and Internal Clock Synchronization.

- External clock synchronization is the one in which an external reference clock is present. It is used as a reference and the nodes in the system can set and adjust their time accordingly.
- Internal clock synchronization is the one in which each node shares its time with other nodes and all the nodes set and adjust their times accordingly.

Cristian's Algorithm is a clock synchronization algorithm that is used to synchronize time with a time server by client processes. This algorithm works well with low-latency networks where Round Trip Time is short as compared to accuracy while redundancy prone distributed systems/applications do not go hand in hand with this algorithm. Here Round Trip Time refers to the time duration between start of a Request and end of corresponding Response.

Code:

ClientNew.java

```
        Clock clientClock = Clock.systemUTC();
        // Looking up the registry for the remote object
            SystemTime stubTime = (SystemTime) registry.lookup("SystemTime");

            // Get current time before calling the server to calculate RTT
            long start = Instant.now().toEpochMilli();

            // Calling the remote method using the obtained object
            long serverTime = stubTime.getSystemTime();
            System.out.println("Server time "+ serverTime);

            long end = Instant.now().toEpochMilli();

            // Calulate RTT
            long rtt = (end-start)/2;
            System.out.println("RTT "+ rtt);

            // Calcuate updatedTime to set the client clock with RTT delay
            long updatedTime = serverTime+rtt;

            // Calculate offset
            Duration diff = Duration.ofMillis(updatedTime - clientClock.instant().toEpochMilli());

            // Set Client clock based on offset to server time
            clientClock = clientClock.offset(clientClock, diff);
            System.out.println("\nNew Client Time "+ clientClock.instant().toEpochMilli());
            // -->
      Scanner myObj = new Scanner(System.in);
      int choice =0;
```

DefaultSystemTime.java

```
import java.time.Instant;

// Implementing the remote interface
public class DefaultSystemTime implements SystemTime {
    // Implementing the interface method
    public long getSystemTime() {
        // Calculating Epoch time on server
        long time = Instant.now().toEpochMilli();
        System.out.println("This is the current time on the server, "+ time);
        return time;
    }
}
```

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);
```

SystemTime.java

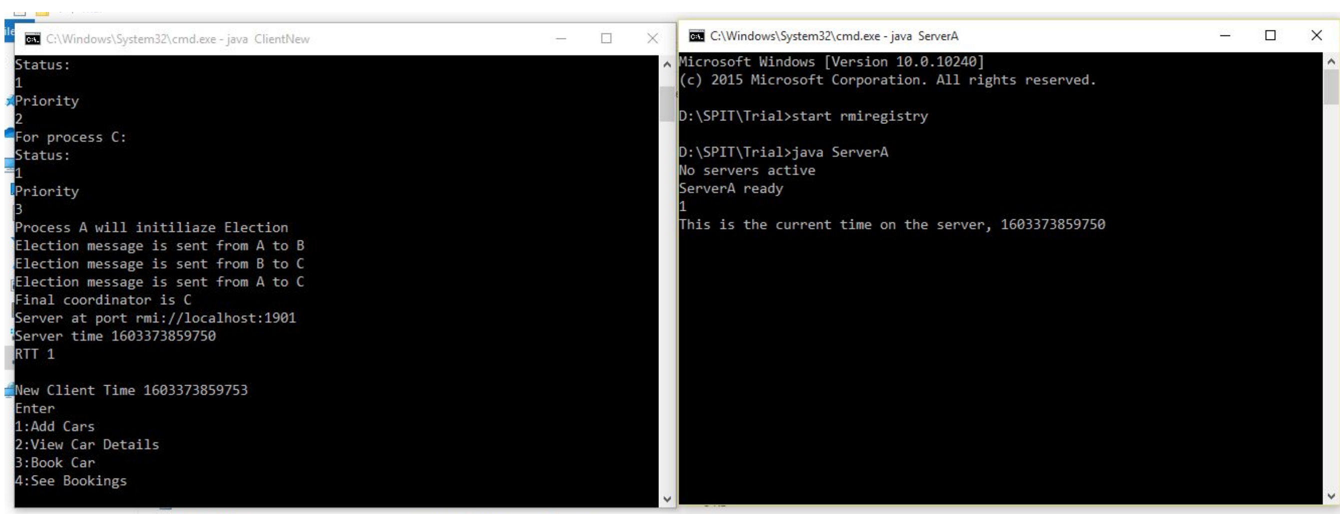
```java
import java.rmi.Remote;
import java.rmi.RemoteException;


// Creating Remote interface for our application
public interface SystemTime extends Remote {
    long getSystemTime() throws RemoteException;
}
```

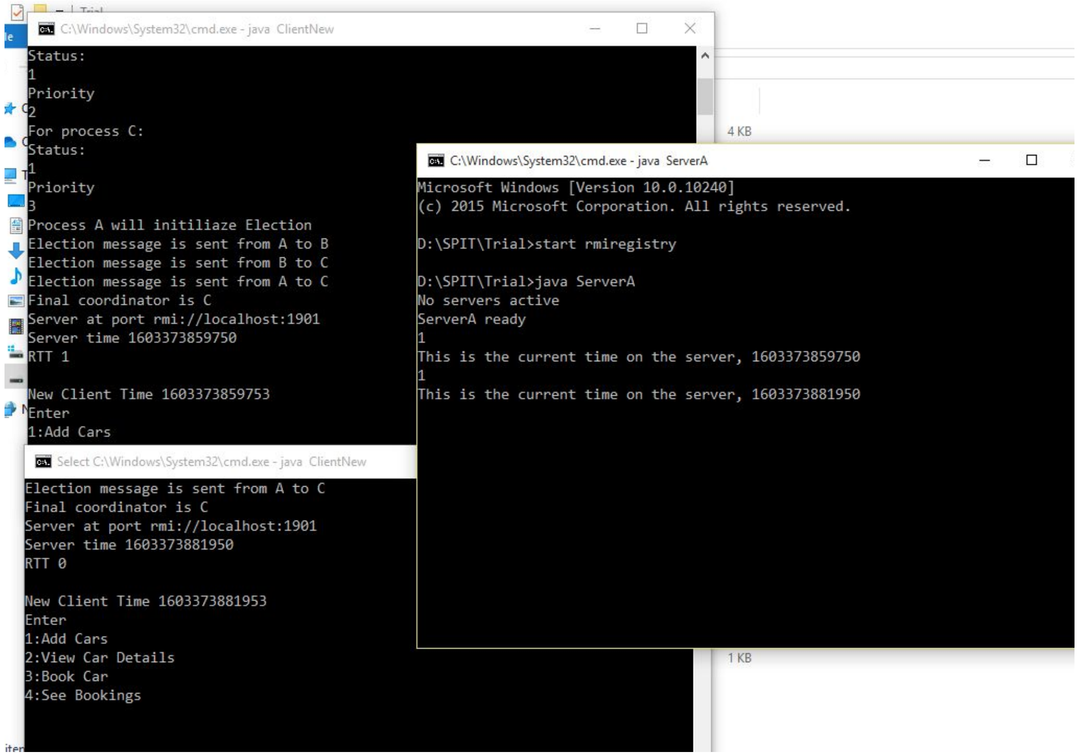
Explanation of Implementation:
When we run ClientNew.java, the process on the client machine sends the request for fetching clock time(time at server) to Clock Server i.e ServerA.java in our system. The Clock Server listens to the request made by the client process and returns the response in form of clock server time. Now the client process fetches the response from the Clock Server and calculates the synchronised client clock time. Basically, the round trip time is calculated. When a second client is started, the same process is carried out for that client. This is how Christian algorithm is implemented in our system.
First the time of the client is calculated. Then we call the server to calculate the round trip time.
rtt = (end-start)/2
The round trip time is calculated with the above formula. Start is the time before calling the server. End is the time after response is received. The client clock is then updated by adding the roundtrip time to the server time. The client clock is then based on the offset to the server time.

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.

Screenshots:



A single client and server is there. The client time and server time is printed respectively.



There are 2 clients and a single server. Both the clients have different times. Each client prints its respective time. The server prints its time.

Conclusion:

We have implemented clock synchronization. The server clock time as well as the client time is printed. Each time a new client connects, the time for that is printed. In this manner, clock synchronization was implemented and integrated in our car booking system.