EE 382N: Distributed Systems
Homework 2

Instructor: Professor Vijay Garg (email: garg@ece.utexas.edu)
TA: Changyong Hu (email:colinhu9@utexas.edu)

Deadline: Oct. 11th, 2019

This homework contains a theory part (Q1-Q2) and a programming part (Q3). The theory part should be written or typed on a paper and submitted at the beginning of the class. The source code must be uploaded through Canvas before the end of the due date (i.e., 11:59pm in Oct. 11th). The assignment should be done in teams of two. You should use the templates downloaded from the course github. You should not change the file names and function signatures. In addition, you should not use package for encapsulation. Please zip and name the source code as EID1_EID2.zip.

1. **(10 pts)** Show that if G and H are consistent cuts of a distributed computation $(E, \rightarrow)$, then so are $G \cup H$ and $G \cap H$.

2. **(10 pts)** Show all consistent cuts of the following computation using the Hasse-diagram.

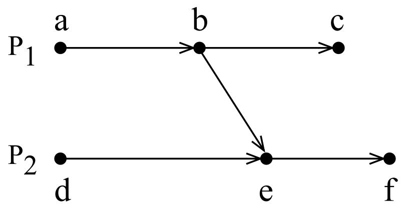

Figure 1: Computation on two processes

3. **(80 pts)** Implement a fault-tolerant client-server ticket reservation system by extending the one developed in previous homework. The fault-tolerance on the server side is achieved by replication. Consistency in replication is maintained by using mutual exclusion for any updates using Lamport's mutex algorithm. Thus, there are n "identical" servers that maintain the current status of all the seats. Any server can communicate to any other server. Your system should give the user illusion of a single immortal server (so far as serving the requests are concerned). A client connects to a server that is closest to it — details on how to determine this server are provided later in the write-up. If the connection is not successful within 100 milliseconds (*you must use only this timeout value in your code*), the client assumes that the server has crashed, and it contacts the next server based on the proximity order. It loops on the server addresses until a successful connection is established. Note that a client always starts with the closest (first) server in its list. You may assume that there is at least one server that is always running. *Do not assume that it is the same server that is always up.* When a server comes up again (after crashing), it would need to synchronize with existing servers to

ensure consistency. Your program should behave correctly in presence of multiple concurrent clients. You only need to use TCP protocol for this implementation.

Input Format: Your program will read input from standard input, and write output to standard output. The server should be implemented in class `Server.java`, and the client class should be named `Client.java`. Note that you must ensure that after the servers/clients have been started with their respective inputs, they can execute concurrently as separate Java processes; i.e. both of these classes should have `main` methods that read from standard input.

The first line of the input to a server contains three natural numbers separated by a single white-space: $server - id$: server's unique id, n: total numbers of server instances, and z: the total number of seats in the theater. The numbers of the seats are defined from 1 to z. The next n lines of the server input define the addresses of all the n servers in the `<ip-address>:<port-number>` format, one per line. The `<ip-address>:<port-number>` of the i-th address line denotes the ip address and port number of server with id i. The crash of a server is simulated by 'Ctrl-C'.

Given that we will only use TCP, the client inputs are slightly different than those in previous homework. A client also accepts its commands by reading standard input. The first line of client input contains the n: a natural number that indicates the number of servers present. The next n lines of client input list the ip-addresses, and port of these n servers, one per line in `<ip-address>:<port-number>` form. Their order of appearance in client input defines the server proximity to this client, and the client must connect to servers in this order.

The remainder of the client input contains seat reservation and return commands that should be executed by the client in order of their appearance. The format of these commands is one of the following:

- `reserve <name>` – inputs the name of a person and reserves a seat against this name. The client sends this command to the server. If the theater does not have enough seats(completely booked), no seat is assigned and the command responds with message: 'Sold out - No seat available'. If a reservation has already been made under that name, then the command responds with message: 'Seat already booked against the name provided'. Otherwise, a seat is reserved against the name provided and the client is relayed a message: 'Seat assigned to you is `<seat-number>`'.
- `bookSeat <name> <seatNum>` – behaves similar to reserve command, but imposes an additional constraint that a seat is reserved if and only if there is no existing reservation against name and the seat having the number `<seatNum>` is available. If there is no existing reservation but `<seatNum>` is not available, the response is: '`<seatNum>` is not available'.
- `search <name>` – returns the seat number reserved for name. If no reservation is found for name the system responds with a message: 'No reservation found for `<name>`'.
- `delete <name>` – frees up the seat allocated to that person. The command returns the seat number that was released. If no existing reservation was found, responds with: 'No reservation found for `<name>`'.

You can assume that the servers and clients are always receive consistent and valid command from users. Here is a small example of the inputs.
Inputs

---

server1 input: **server2 input**:

```
1 2 10
127.0.0.1:8025
127.0.0.1:8030
```

client1 input:

```
2
127.0.0.1:8025
127.0.0.1:8030
reserve Bob
delete Mary
search John
```

```
2 2 10
127.0.0.1:8025
127.0.0.1:8030
```

client2 input:

```
2
127.0.0.1:8030
127.0.0.1:8025

bookSeat Alice 8
```