EE 360P: Concurrent and Distributed Systems
Assignment 4

Instructor: Professor Vijay Garg (email: garg@ece.utexas.edu)

Deadline: March 21, 2017

The source code (Java files) of this assignment must be uploaded through the canvas before the end of the due date. The assignment should be done in teams of two. You should use the templates downloaded from the course github (https://github.com/vijaygarg1/EE-360P.git). You should not change the file names and function signatures but you can add other java files. In addition, you should not use package for encapsulation. Please zip and name the source code as [EID1_EID2].zip. You should include all your java files.

1. **(100 pts)** Implement a fault-tolerant client-server online store system by extending the one developed in assignment 3. 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 the online store. 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. You can assume that a server will respond within 100 ms if it is alive. The client 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 the majority of the servers do not crash. Once a server has crashed, it stays crashed. You do not have to worry about the server joining the group after a crash. 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 two natural numbers separated by a single white-space: `server-id`: server's unique id, `n`: total numbers of server instances, and a string `inventory-path`: the file path indicating all products of the store. 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 assignment 3. 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 commands that should be executed by the client in order of their appearance. Every client accepts only the following commands from standard input:

(a) `purchase <user-name> <product-name> <quantity>` – inputs the name of a customer, the name of the product, and the number of quantity that the user wants to purchase. If the store does not have enough such items, the server responds with message: 'Not Available - Not enough items'. If the store does not have the product, the server responds with message: 'Not Available - We do not sell this product'. Otherwise, an order is placed and the server replies a message: 'You order has been placed, `<order-id> <user-name> <product-name> <quantity>`'. Note that, the order-id is unique and automatically generated by the server. You can assume that the order-id starts with 1. The server should also update the inventory.

(b) `cancel <order-id>` – cancels the order with the `<order-id>`. If there is no existing order with the id, the response is: '`<order-id>` not found, no such order'. Otherwise, the server replies: 'Order `<order-id>` is canceled' and updates the inventory.

(c) `search <user-name>` – returns all orders for the user. If no order is found for the user, the system responds with a message: 'No order found for `<user-name>`'. Otherwise, list all orders of the users as `<order-id>, <product-name>, <quantity>`. Note that, you should print one line per order.

(d) `list` – lists all available products with quantities of the store. For each product, you should show '`<product-name> <quantity>`'. Note that, even if the product is sold out, you should also print the product with quantity `0`. In addition, you should print one line per product.

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

Inputs

---

server1 input:

```
1 2 inventory.txt
127.0.0.1:8025
127.0.0.1:8030
```

server2 input:

```
2 2 inventory.txt
127.0.0.1:8025
127.0.0.1:8030
```

client1 input:

```
2
127.0.0.1:8025
127.0.0.1:8030
purchase Tim phone 10
list
```

client2 input:

```
2
127.0.0.1:8030
127.0.0.1:8025
search John
search Tim
```