Assignment # 2

Due Date: Oct 22 (Thu)
Package the source code in a zip file, and email (put [CSE 535] Assignment2 submission in subject line)

1. **(60 points)** Implement a fault-tolerant library that has unique natural numbers as book ids. The fault-tolerance on the server side is achieved by replication. Consistency in replication is maintained by using mutual exclusion for any updates using Lamports mutex algorithm. Thus, there are n identical servers that keep the current allocation of the books. All servers know ip addresses of other servers. Your system should give the user illusion of a single immortal server (so far as serving the requests are concerned). A client may connect to any of the servers. Assume that a client chooses a server at random. If that server is down (i.e. after the timeout), the client contacts some other server. 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, it would need to synchronize with existing servers to ensure consistency. You can assume perfect failure detection. Your program should behave correctly in presence of multiple concurrent clients.

   The server accepts only the following calls from a client terminal:

   - `<clientid> <booknumber> reserve`. If the book is issued to someone else, or does not exist the in the library, the call client outputs: `fail <clientid> <booknumber>`, otherwise the client outputs `<clientid> <booknumber>`.
   - `<clientid> <booknumber> return` – If the book with the given number was issued to the `<clientid>`, then it is returned to the library, and the client outputs: `free <clientid> <booknumber>`, in all other cases the client outputs: `fail <clientid> <booknumber>`.

   Here `<text>` is used to indicate the value of the parameter denoted by `text` in the given context. The assignment requires you to create both server and client programs, such that the client (processes/threads) communicate with any server using sockets. Your program should behave correctly in presence of multiple concurrent clients.

   Input Format: Your program will read input from standard input/file, and write output to standard output. There is an input file for each server, as well as for each client. Note that you must ensure that after the servers/clients have been started with their respective input files as arguments, they execute concurrently as separate Java processes. For servers, each input file has the same information: first line of the input contains 2 non-negative integers n, and z that respectively specify the number of server instances, and the total number of books in the library. The ids of the books are defined as $b1, b2, \ldots bz$ and the ids of the servers are defined as $s1, s2, \ldots sn$. The next n lines in define the address of the server in the `<ipaddress>:<portnumber>` format. If the `<ipaddress>` corresponds to the localhost, then your program should start a server on the given `portnumber` (using a Server Socket) for each such configuration. If the `<ipaddress>` does not correspond to the machine the program is being executed on, then assume that a server is already present and running on that corresponding address (and the port). Rest of the input defines the commands passed to server in the manner of one command per newline. The commands take the form:

   - $si\ \ k\ \ \Delta$
     where si is the server id ($1 \leq i \leq n$) that gets assigned to the server based on the input file it is parsing, k is a non-negative integer that tells the server to become unresponsive after receiving k^{th} message

(from the beginning of its latest start; and irrespective of message origin - be it a client or a server - but not counting the socket connection open/close messages), and Δ is the duration (in milliseconds) for which si would become unresponsive – to all clients and servers.

For client input files, the first line of the file contains one non-negative integer n that indicates the number of servers present. The next n lines list the ip-addresses of these n servers, one per line. *The client should attempt connections to servers (using these ip-addresses) in this order specified by line breaks.* The lines $n+2$ to end-of-file contain the commands the client has to execute, and all of them start with `ci` - the id assigned to the the client. Hence, the id of the client is defined by the input file it is given to parse and execute. The format of the commands listed in lines $n+2$ to end-of-file will always be one of the following:

- ci bj `reserve|return`
  where ci is the client id ($i \geq 1$). Details of these commands were defined above. Note that each client is its own independent process/thread, which should be started by you, and executes concurrently. Thus `reserve c1 b1` followed (without significant delays) by `reserve c2 b1` does not mean that it is guaranteed that `b1` must always be issued to `c1`.
- ci Δ
  Δ is the time (in milliseconds) for which the client would become dormant (using sleep), immediately after receiving the command.

Output Format: The result of each client command is printed in a separate client output file, with the format of the client output described above, one per line, with a single blank separating every alpha-numeric word.

Here is a small example that you can easily verify by hand.

Inputs

server1.in:
```
2 10
127.0.0.1:8025
127.0.0.1:8030
s1 3 3000
```

server2.in:
```
2 10
127.0.0.1:8025
127.0.0.1:8030
```

client1.in:
```
2
127.0.0.1:8025
127.0.0.1:8030
c1 b8 reserve
c1 b2 return
c1 b2 reserve
```

client2.in:
```
2
127.0.0.1:8030
127.0.0.1:8025
c2 1500
c2 b2 reserve
```

Outputs

c1.out:
```
c1 b8
fail c1 b2
c1 b2
```

c2.out:
```
fail c2 b2
```