**Terri-Anne Cambridge 9373071**

SOEN 423, FALL 2011

Assignment 2

October 31, 2011

Distributed Ticket Reservation System (DTRS) using CORBA

This document is an adaptation of the previous assignment on Java RMI to the current assignment on Corba.

# Techniques

In order to fill the Box Office with shows, I applied a simplified version of GoF’s Fabrication pattern. It is a singleton class whose sole purpose is to return a list of instantiated shows. It creates three shows and sets its show number and number of tickets.

The CustomerClient class inherits from the Thread class. This allows me to process each client as a thread and have them accessing the box offices, reserving and canceling tickets concurrently.

To store customer information and show information, I initially set out to use HashMaps. It is not synchronized and multiple threads can access it at the same time. However, if a thread modifies the HashMap by adding or deleting an entry while it is being operated on by another thread, it can cause problems.

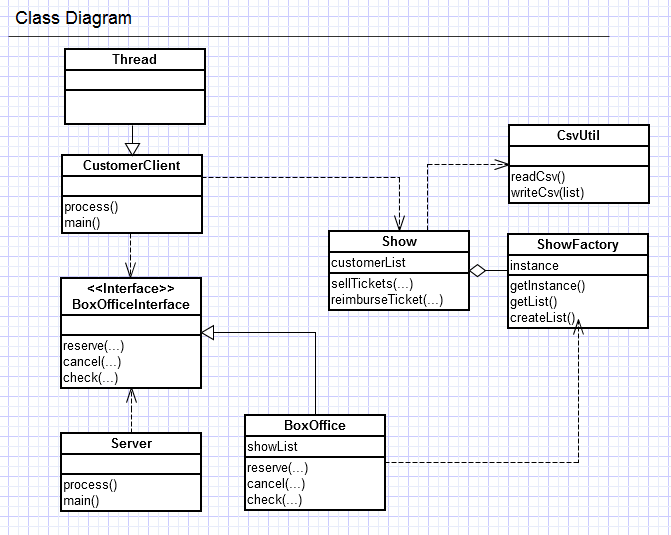
I then considered using its alternative, which is the ConcurrrentHashMap implementing ConcurrentMap. Rather than locking the entire map, it implements a concurrencyLevel, setting it to 16, by default, which means that 16 threads can access the map at the same time, as long as they are accessing different elements. It thus reduces the need for synchronization and allows for iteration on the map while being modified.

# Architecture

The application is split into three parts: the server layer servicing the client, the client invoking server methods, and the common package containing shared components. That is further split into three parts: local Show class, remote BoxOffice class, and a CsvUtil class used to read and write to a file.

# Class Diagram

The following is a simplified class diagram representing the application. Details were omitted for simplicity to emphasize and simply convey the organization of the code.



In order to adapt to Corba and the new requirements, the BoxOfficeInterface class was removed, and a udp package was added containind UDPServer, UDPClient, Marshaller which implements methods to marshall and unmarshal a data structure, and IUpdatable which is an interface to be implemented by objects that will be sent through the sockets. It contains a method signature for updating when handled by the server. Methods “exchange(..)” and “canExchange(…)” were added to BoxOffice in order to perform the exchange operation. An Exchange class, which implements IUpdatable, was also added and serves as an exchange message passed through the socket. Additional classes generated by the CORBA plugin were also added but are not listed here.

# Testing Scenarios

I made use of Java’s unit testing tool, JUnit. I used this in order to test the services provided from the BoxOffice Class; namely reserve(…), cancel(…) and check(…).

My Test cases are as followed:

/\*

\* Reserve ticket

\* Expected Result: the ticket is reserved.

\*/

**public** **void** testReserve() {..}

/\*

\* Reserving more tickets than available

\* Expected result : no tickets are reserved

\*/

**public** **void** testReserveAboveQuantity() {..}

/\*

\* Cancel ticket

\* Expected Result: the ticket is canceled.

\*/

**public** **void** testCancel() {..}

/\*

\* Canceling more tickets than user bought

\* Expected Result: Only the tickets reserved are reimbursed, not more.

\*/

**public** **void** testCancelAboveQuantity() {..}

/\*

\* Check number of available tickets after reservation

\* Expected result: the correct number of available tickets is returned.

\*/

**public** **void** testCheck() {..}

I also tested the CsvUtil functionalities. The test cases are provided below:

/\*

\* Write customer list to file

\* Expected Result: all entries are written

\*/

**public** **void** testWriteOneRecordPerLine() {..}

/\*

\* Read customer list from file

\* Expected Result: all lines are read

\*/

**public** **void** testReadOneRecordPerLine() {..}

I also manually tested how the threads interacted and modified the data. This was accomplished by debugging in order to correct areas of inconsistent data. An example is when a customer purchases tickets, but this purchase is not reflected in the log files.

# Important Aspects and Difficulties

Difficulties encountered were mostly due to particularities of CORBA. One of them was trying to find a way to describe a method in IDL which would return a Map. I tried using Any, Sequence, or Array but in the end did not satisfy what I needed therefore I resorted to other means. Since I only needed such a method to return a list of shows in order to print their data, I could simply store the data in a string and return that instead from the server.

Another difficulty encountered was setting up communication between servers. Getting the target server from the calling server and then passing it to the exchange message to perform its operations was causing errors, particularly “delegation not set”. I then resorted to get the target within the exchange class rather than passing it as parameters, and this proved to work with no problem.

Implementing the UDP Client and Server was very simple and with the example provided from the generated files, it made the development go a lot smoother.

Though last time, the complexity was mainly focused with implementing threads and concurrency, having the logic done already, made it a lot easier this time.

Taken from the documentation produced for the Java RMI application:

*The real difficulty was when implementing threads and concurrency. I had to test and debug to solve the issue where an update was lost, as mentioned above. For example, a customer purchasing a ticket, but the logs written would not show this. Therefore there was a synchronizing issue in the reads and writes to the file. Other similar issues arose and root causes were found through debugging.*

Atomicity for the exchange operation is ensured with the synchronized keyword, so that only one thread can perform this operation on a box office at a time. Not only so, but this operation first checks if the other box can reserve the desired tickets. If so, it will perform the reservation and a message will be passed to the current box office to cancel its reservations. If not, no tickets will be canceled. It’s done in this order because canceling will not fail. If a customer requests to cancel more tickets than they reserved, no exception is thrown because it’ll only cancel up to the amount that the customer reserved. That being said, if the order of inner operations was changed such that the cancel operation was performed first, then the reservation for the desired show, and then the reservation failed, it would require a rollback on the cancelation, which would mean re-reserving the lost tickets. But that may fail if another customer purchased the last set. Therefore, the attempt to reserve from the desired box office should be done before canceling the tickets from the current show.