CMSC433 Project 4: Distributed E-book library with RMI

Due April 25th (Friday), 11:59:59pm

Overview

For this project you will be implementing a distributed E-book library that will use Java RMI to allow members to check out library resources. The goal of this project is for you to better understand distributed object computing using Java RMI.

Getting Started

Download the skeleton files from the project webpage. Read the following sections of this document and all comments in the skeleton files to get a better understanding of the functionality you'll need to implement.

The Project Setup

The project consists of two packages (library and member).

The library package consists of the LibraryServer interface which defines the basic functionality of the server. There is a LibraryServerImpl class which you will complete to implement that interface. This package also consists of a Book class which is Serializable and is one of the data objects that is passed between the client and server.

The member package consists of the Member interface which defines the basic functionality of the client. There is a MemberImpl class which you will complete to implement the interface. This package also consists of a MemberData class which is Serializable and is one of the data objects that is passed between the client and server.

Do not change the class names of the files provided or the current method names, constructors, or parameters as we will be using them to test your code. You may add any other classes, methods, constructors, or variables as you need.

The project also includes a security.policy file to allow RMI. Do not modify this file.

General Requirements

The library is the server that maintains a list of books and registered members.
Books are created based on the arguments passed to the LibraryServerImpl constructor: numBooks (the number of books in the library) and copiesPerBook (copies per book). LibraryServerImpl creates numBooks with unique titles that you should come up with. One way of doing so is to have a prefix constant string such as "Book" and a static counter. This would

give book titles of the form “Book1, Book2, Book3, … Book<numBooks>. You may use other prefixes or create unique book titles in another way.

The library server checks each transaction for the following properties:

- there is a finite number of copies per book
- there is a limit on the number of books that can be checked out by a member
- members cannot check out multiple copies of the same book
- non-members cannot check out books

Members are the clients. Members can do the following transactions. They can:

- register for a membership in the library
- check out books
- return books
- maintain a list of books they have checked out
- maintain a list of books they have read

Update: the books checked out list is the list of books currently checked out by a member. When a member checks out a book successfully, it is added to the checked-out list, and when a book is returned, it is removed from the checked-out list.
The books read list consists of the names of the books a member has read. When a member returns the book, we should assume that the member has finished reading it and it should be added to the books read list. It is a running history of what the member has read.

For this project, two members cannot have the same name.
All member states should be consistent with the server’s state and vice versa.
Update: This means that the library should also have its own record keeping of the the books checked out by each member. The library should not rely on the member data objects passed in to make decisions. It should rely on its own internal records to make decisions. The reason is that a member may “lie” or send erroneous MemberData. In fact, we will be testing these cases.

You **must** ensure that your implementation of your server can handle multiple clients and keep their state consistent with that of the server. Ex. if there are multiple members trying to checkout the same book and the library has a fixed number of copies, n, of that book, then no more than n requests can be honored at any one time.

Implementing RMI

The LibraryServerImpl and MemberImpl are the implementations of the server and client, respectively. Both the server and client will need to be remotely accessible. Implement RMI as covered in lectures.

Running RMI

For the sake of simplicity, you will run all of the elements in different JVMs on a single machine. However, in practice, the elements would potentially be running on different hosts.

Run the main method for LibraryServerImpl which should start the RMI registry and register the server Object. Next run the main method for MemberImpl. The MemberImpl will interact with the LibraryServerImpl using RMI.

Testing

You should create test cases to test the functionality of the client and server thoroughly.
Update: As mentioned in section Running RMI, you should test the RMI functionality through the main methods of the LibraryServerImpl and MemberImpl. This includes the registry creation, the registry binding, and all further communication between the library and member remote objects.

Our testing will ignore your main methods and RMI setup. We will have our own testing classes that will create the registry and library and member objects and will call the appropriate methods for each using RMI. For this reason, all of your registry creation should be in the main method and no where else, so that it does not interfere with our setup. In addition, we will test your LibraryServerImpl and MemberImpl independently in isolation.

What To Turn In

Every file you submit should have your name and UID. To enforce academic integrity, code will be checked for similarity to other submissions. Each student should make his or her own individual submission. Use Eclipse to submit your project or submit a zip file to the submit server directly.

Recommended References

An Overview of RMI Applications - http://docs.oracle.com/javase/tutorial/rmi/overview.html
Lecture slides and examples