Electronic Commerce System
Jian He and Faria Khalid

Date: 7th December 2000

Abstract

The aim of the project is to develop a better understanding of implementation aspects of operating system concepts, especially in distributed object applications. This project familiarized us with Remote Method Invocation (RMI) mechanisms in Java.

The system consists of four different “shoppers”, two "stores", and a "bank." The system allows the shoppers to view and purchase items from both stores. As part of the purchase order, the shopper provides an "account number" and other related information, which the store will verify by contacting the "bank". Two-Phase commit protocol is applied at the confirmation phase of the project. The interaction between the shopper and the store is asynchronous. That is, the shopper does not block while waiting for the store to reply to its requests to view or purchase items. This allows the shopper to conduct business with both stores simultaneously.

Some important concepts are involved in building this system. It is implemented as an object model, which requires synchronization to avoid deadlocks and data inconsistency. To establish communications between objects over network, references of remote objects can be obtained by RMI mechanisms. A “bank” and two “stores” manage transactions in the aspects of Event ordering/Scheduling and Fault Tolerance. To protect the remote objects, a security manager supplied as part of the RMI system is set up for both server and client. It enforces a security policy in what access on objects is allowed.
1. System Architecture

Functionally, the system is made of three parts that communicate with each other and act as client and servers at different steps of the process.

1. A bank server. It is responsible for account management and final confirmation of purchase. The clients of the bank server are stores.

2. Two store servers. One is a Bookstore and the other is a Gift store. They maintain the consistency of inventory; manage transactions and issue the first phase of confirmation with shoppers. The clients of the store server are shoppers.

3. A shopping server for each shopper. It creates two threads, each interfacing with one store. Threads provide a user interface to get item list, select desired item and place an order. The clients of the shopping server are bank and stores when they are seeking confirmations with shoppers.

For each pair of server/client, the server calls the RMI registry to associate a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. Figure 1-2 (copied from <java.sun.com>) shows that the RMI system uses an existing web server to load byte codes of classes written in the Java programming language, from server to client and from client to server, for objects when needed. RMI can load class byte codes using any URL protocol (e.g., HTTP, FTP, file, etc.) that is supported by the Java platform.
When the system is started, all the servers are made to run in a certain order.

1. RMI Registry
2. Bank Server
3. Two Store Servers
4. Shopping Servers

First, all the servers register their services in RMI registry associated with their URL.

Secondly, the bank server prepares for further account operations as the commerce system receives purchase requests from shoppers. The store server needs to look up bank service and obtain the bank manager. Its following tasks are preparing inventory and responding to transaction requests.

When the shopping server is started, it shows a user interface with buttons to enter both stores. Buttons can be clicked once to create a thread interacting with the specified store. This is to ensure that only one thread for the same store is active at the same shopping server. Otherwise, stores would be confused about which shopper of these threads needs to confirm with. Same concern is ensured in some buttons of thread frames. It is not necessary that shoppers can only shop at the same shopping server. Thread frames can accept any shopper’s requests and verify information by contacting stores.

2. System Functions

Shoppers can send requests to either store via the corresponding thread frame.

1. View the list of items.

2. Order an item. After verifications on accounts and other information, the store will reserve one item for the shopper. The button “order” is disabled until the shopper’s confirmations are done. It also locks the text input fields like item number, etc. to prevent them from being changed before confirmation.
3. Confirm the order. This will activate bank confirmation frame for the final purchase confirmation. Confirmation is implemented with two-phase commit. First, the store seeks confirmation from the shopper about the purchase and then, sends this information to the bank for money transaction. Before authorizing the expenditure, the bank seeks confirmation from the client about the amount and the store to see if they are acceptable.

4. Cancel the order. It makes the store put back the reserved item same as bank confirmation frame does with cancel request.

5. Exit the store. It sets the thread frame invisible and enables the entry button for the specified store.

4. **UML**

<table>
<thead>
<tr>
<th>BankManager</th>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account(String accountNumber): Account</td>
<td>+getBankManager():BankManager</td>
</tr>
<tr>
<td>getClient(String clientName): ClientStore</td>
<td>+getClient():ClientStore</td>
</tr>
<tr>
<td>transferAccCon(amount: long, ShoppingHost: String, ShoppingPort: String, ShoppingService: String, m_Store: String, src_acc: String): int</td>
<td>+getBalance():long</td>
</tr>
<tr>
<td>transferAcc(amount: long, dest_acc: String, src_acc: String): boolean</td>
<td>+getCash(amount: long): long</td>
</tr>
<tr>
<td>deposit(amount: long)</td>
<td>+deposit(amount: long)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StoreManager</th>
<th>ClientStore</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item</td>
<td>+getBankManager():BankManager</td>
</tr>
<tr>
<td>+getItem(itemNumber: String): Item</td>
<td>+getName():String</td>
</tr>
<tr>
<td>+orderItem( ShoppingService: String, ShoppingHost: String, ShoppingPort: String, itemNumber: String, shopperName: String, accountNumber: String): int</td>
<td></td>
</tr>
<tr>
<td>+getItemList():Hashtable</td>
<td>+getPrice():long</td>
</tr>
<tr>
<td>+getSold( itemNumber: String, accountNumber: String): int</td>
<td>+getQuantity():int</td>
</tr>
<tr>
<td>+cancelOrder(itemNumber: String)</td>
<td>+getType():int</td>
</tr>
<tr>
<td>+finalSold(itemNumber: String, accountNumber: String): int</td>
<td>+sellitem(number: int)</td>
</tr>
</tbody>
</table>
There are three packages: bank, store and shopper. Each includes the implementation of a server and a client. In packages, a class or an interface may import the other two packages when it needs the class definitions from them to generate remote object references at different stages. These packages contain interface definition (Object.java) and interface implementation (ObjectImpl.java) for remote objects exported via RMI. Beside remote objects, packages contain local server programs (*Server.java) and exception class definitions.

1. Package bank
   It contains:
   Three remote objects: BankManager and Account Object
   Object Implementations: BankManagerImpl, and AccountImpl
   The bank server: BankSystemServer.
   Exception class: NoCashAvailableException

2. Package store
   It contains:
   Two remote objects: StoreManager, Item and ClientStore.
   Object Implementation: StoreManagerImpl, ItemImpl and ClientStoreImpl
   The store server: StoreServer.
   Exception class: NoItemAvailableException

3. Package shopper
   It contains:
   One remote object: ShoppingManager.
   Object Implementation: ShoppingManagerImpl.
   The shopping server: StoreShopperApplet, ShopperThrdFrame and ShopperConFrame

The relationships between these packages are shown in Figure 4-2.
Remote Interfaces:  

Local Servers/UI:

![Diagram showing relationships of packages](image)

Figure 4-2 Relationships of Packages

5. Summary

This project gave us a good hands-on about distributed object applications. By programming and doing experiments, we learned not only Java RMI techniques, but also some important considerations in distributed system design, such as synchronization, event ordering etc., as mentioned in abstract.

It’s valuable to study RMI stub and skeleton mechanism, which provides programmer a standard way to build communications with remote objects. We did not build any threads for servers. Because the stub hides the serialization of parameters and the network-level communication, invocations by clients are actually serialized by RMI stubs on server side.

At very beginning, we did not use synchronization. During experiments, the whole system with multiple shoppers did not experience any data consistency without synchronization. However, a method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no
guarantees with respect to mapping remote object invocations to threads. Thus, we use “synchronized” to control access to shared objects, like Hash tables which serve as simple database system for bank accounts and inventory of stores.

There are some extensions that can be made with the project. Database connectivity would help servers recover all the information from system failure. Also, it would be nice if the store could show the images of the item selected by shoppers. Adding cryptography in exchanging information between server/client can enhance system security. In the real world e-commerce system, it’s a very important issue to protect account information.

All in all, it’s a good project.