ELECTRONIC COMMERCE SYSTEM USING JAVA RMI

MEMBER:

Ming LUO(990-22-4595) and **Jun WANG**(900-20-6643)

ABSTRACT:

Electronic Commerce is a very hot and important area in Internet Applications. It is one of the major motivations for the development and implementation of most famous commercial web sites. In this project we use Java RMI(Remote Method Invocation) to implement a simple Electronic Commerce System which includes Clients, Stores and Bank. Graphic User Interface is provided in Client site on the basis of Java SWING package. The number of Clients and stores can be added without changing any source code. Store and bank act as RMI server, while Client uses callback technique to avoid possible nested RMI call and provide a more practical way of implementation. The core concept of this project is invoking methods implemented remotely and using nonblocking multi-thread archetecture, so asynchronized purchasing and confirming is allowed in this system. Also this project gives us an opportunity to abtain insight of designing distributed systems, such as remote access, concurrent events, nonblocking calls and Java security policy.

MOTIVATION

There are huge demands of Electronic Commerce in a current networked world. You can buy and bid lots of stuff in the net. As computer major students we would like to see what is happening under hood of the Ecommerce system. Java language has the advantage of platform independence and web browser integration. With RMI(Remote Method Invocation) we can develop distributed system easily. This project is a simplified Ecommerce system to let us understand deeply about the mechanism of remote calling in a distributed system.

SYSTEM DESCRIPTION

2.1 Introduction

This system consists of Clients, Stores and Bank. Store and Bank act as RMI server while Client use callback technique to avoid possible nested RMI call. Each complete purchase activity involves Purchase Request(Client to Store), Confirm Request(Store to Bank), Callback for Verify(Bank to Client), Giving Verify Result(Client to Bank), Giving Confirm Result(Bank to Store) and Giving Information Back To Client(Store to Client).

2.2 Structure

Here is the abstract picture on the way the system works.

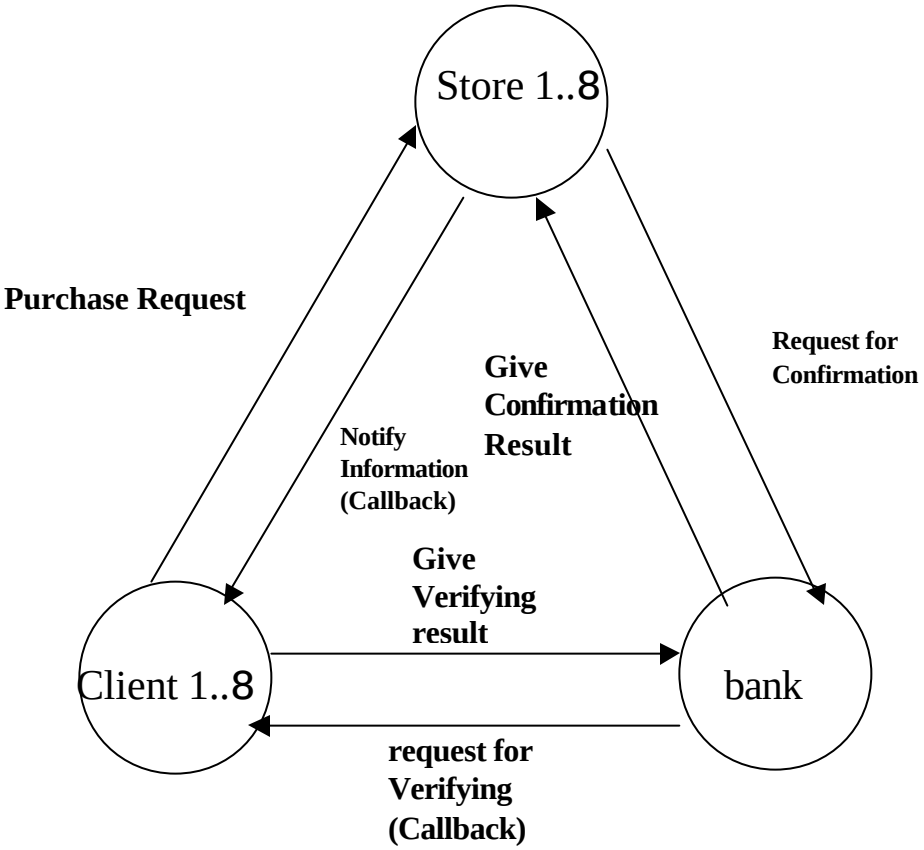

Fig 1. System Architecture

RMI Interface Definition:

On Client site there is no RMI server. We designed this architecture for two reasons. One is to avoid the possible nested call of RMI. Figure 2 shows the architecture of three servers architecture.

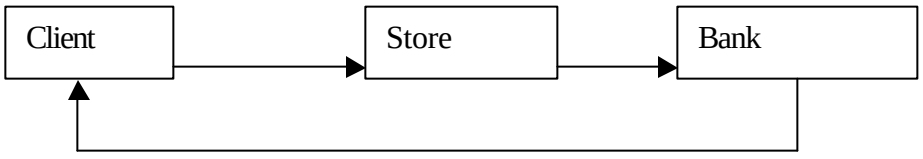

Fig 2. Three Servers Architecture

This does not mean using three servers model will unavoidably cause this problem, but we do agree that it is more probe to introduce defects and we think our architecture is safer and more delegate. The other reason we use call back is that it is unnatural to require the client to start a server when he/she wants to buy something in the net. So our architecture is more natural. Figure 3 shows how the Callback is transported to the two servers. The Callback actually is an interface for client implementation package. Using this interface, other objects (in this case store and bank) can access client only via predefined functions. After it is passed to Store and Bank, interact with client in a predicable way.

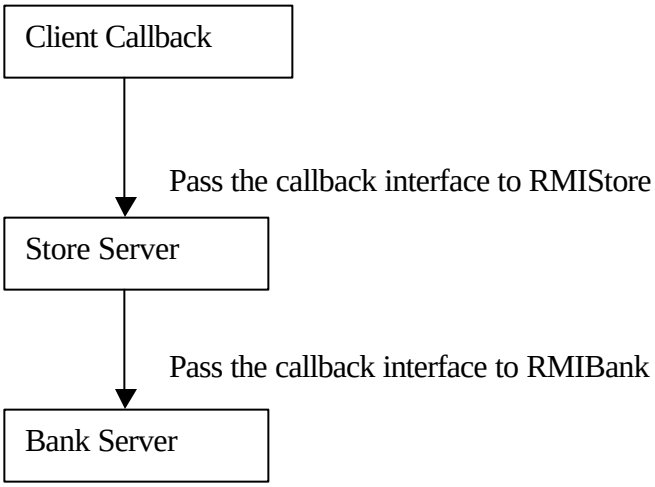

Fig 3. Callback Interface transporting

Here is the source code of all the interface definitions:

```
package commerce;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Callback extends Remote
{
        //called by bank
        /**********************************************************************
        * confirm this customer spending this amount of money using this account    *
        * in this store.                                                            *
        **********************************************************************/
        public boolean confirm(String acct, String sname, double amt) throws
RemoteException;

        //called by store
        /**********************************************************************
        * return the result of this purchase                                        *
        **********************************************************************/
        public void getNotify(boolean b) throws RemoteException;
```

```
}


public interface RMIStore extends java.rmi.Remote
{
        //called by client
        /**********************************************************************
        * return a list of goods, including item id, item name, and item price*
        **********************************************************************/
        public Vector view()
                throws java.rmi.RemoteException;
        /**********************************************************************
         * return purchase success or failure                                 *
        **********************************************************************/
        public void purchase(String acct, //client account
                Vector items, //including gid & qty
                Callback client)        //pass it to the bank
                throws java.rmi.RemoteException;

        public String getName() throws java.rmi.RemoteException;     //just used for GUI
}

public interface RMIBank extends java.rmi.Remote
{
        //called by store
        /*********************************************
         * get the confirmation from the client,     *
         * authorize or deny the expenditure.        *
         *********************************************/
        public boolean verify (String sname, //store name
                String acct, //client account
                double amt, //money amount
                Callback client) //call its verify
                throws java.rmi.RemoteException;
}
```

Implementation:

Client Site:

We construct a Graphic user interface to provide a friendly Human Computer Interface.

User can choose one store among four stores(Paradise Bird Electronic Card Store, Oasis CD shop, Ubid and 800.com). For security reason the 8-digit account number is displayed as "*********". After user clicked Ok button, client can view the inventory from the choosed Store RMI Server by invoking the remote method of "view()" and display it as figure 4.

Then user can choose the items from a list of the goods the store have. The initial quantity for each item is 0. After user checked the "Choose CheckBox", the default value will be automatically changed to 1. User can change Quantity of goods. As showed in the picture, the quantity of index 100006 has been changed to 10. User can choose "reset" to clear all the input. After user clicks Buy Button, a dialog for confirmation is displayed. The total amount is also displayed as in figure 5. At this point, client can check his/her buying list, decide to buy the stuff listed or just cancel this purchase activity and return to the view interface.

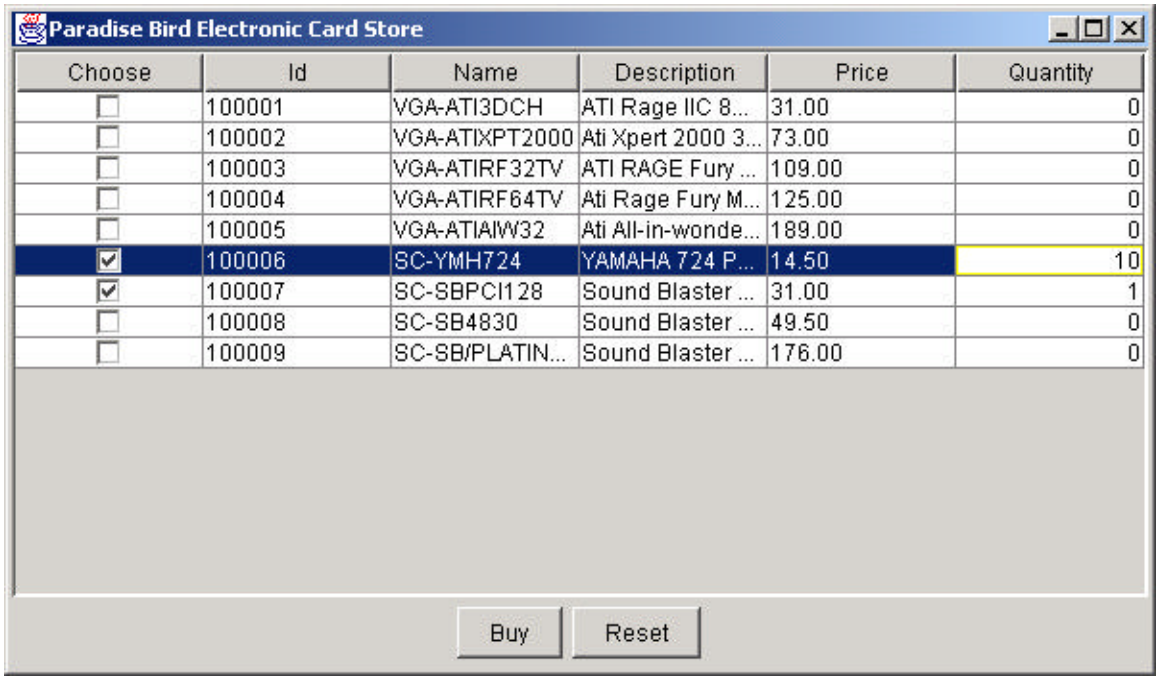

Fig 4 Graphic User Interface--Item Inventory

When client chooses "OK" button (figure 5) on confirmation dialog, a thread is started on client site to handle the following activities. This thread implements the Callback interface, shows a status dialog to indicate the on-going process of purchasing and later accepts getNotify() call from bank and gets purchase success/failure info from store. We pass the Callback interface for this purchase thread to store and bank, so bank and store can call back to client for confirmation and notifying.

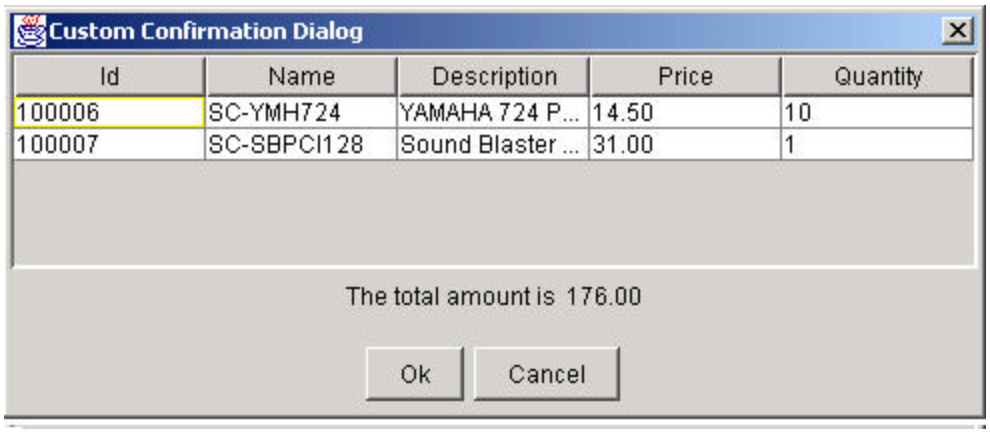

Fig 5 Customer Confirmation Dialog

Store Site:

Store preserves the information of inventory in a file (can be upgraded to a database if we can solve the database policy problem in the graduate lab). RMIStore receives the view request from Client site and returns inventory information. After client sends out the Buy Vector (the list of buying things) by calling RMIStore.purchase(), Store starts a thread which computes the total amount and delivers it to the bank together with the and the client account and the Callback interface it gets.

Why we use multi-thread architecture on store side? Since different stores actually are distinct RMIStore servers, which means they react the same behavior toward the same external stimuli. So we only need to consider the situation of purchases happened in the same store. If the store waits for bank verification and bank waits for client confirmation in one purchase, the same store can not process other clients' requests. To allow queries and purchases happening concurrently, we decide to apply multi-thread architecture. One thread processes only one purchase from the client site. So there is no blocking problem in our system.

Bank Site:

Managing user account information and asking client for confirmation toward one certain purchase are the main tasks of bank server. Bank will determine a user account is valid or not according to user profiles. If it is valid, is there enough money on it? If both cases either the account is not valid or there is not enough money on it the bank will return failure information to store and as a result store will inform client that the transaction fails.

Client can choose to confirm one purchase or not. The bank will return corresponding result to store due to client's response. And there will be the corresponding changes on user account information (e.g. the amount of money in this account) if client choose to confirm.

Figure 6 shows a complete procedure of a successful purchase.

SUMMARY:

The goal of this project is to implement a distributed system using Java RMI. We now understand deeper about the mechanism that RMI works. We find out through practice that RMI gives us a feasible way to invoke methods implemented in remote machines, and that we need to add more operations and serious considerations of things like stub, policy etc. besides operational part due to remote communications. However, a distributed system can be implemented easily in Java, and the attribute of platform independency brought in by Java language makes it not difficult to transport the whole system. Actually this project is first developed on the Unix platform, and later we migrate the client part to Windows platform (due to the machine limitations in graduate lab).

We have a deeper understanding of the concurrent issue and deploying Threads now. During our developing process several traps did show up. At first a store server might be blocked when it was waiting for some response from bank. Later we introduce threads to our system to avoid blocking. There are a bit ponders on data consistency too. We decide after carefully comparing several mechanisms we can think out that preserving account information only on bank site and making every decision about account by bank is a preferred idea to ensure client data consistency.

This system still has much room for extensions. At the beginning we thought of using JDBC to preserver inventory information on store side and client account information on bank side (now we are using files). But we tried and failed because we have no Administrator privilege for accessing the database we set up. About introducing more banks issue (since now our system has already implemented random numbers of clients and stores interaction without any need to modify source code, only by adding some data files for stores and account entry for bank profile), there are several ways. One way is adding a proxy to decide which bank the store should look up for from the account number. Stores can send its request information to this proxy center and this proxy routes the request to certain bank.