# Design Document 423 Assignment 1

# RMI Distributed Retail Server

Jeremy Pallats

ID: 9619550

October 14th, 2012

# Introduction

The goal of this assignment is to make a distributed retail server using the RMI technology and achieve a high degree of concurrency. In order to do this, I've created three Eclipse projects that together implement the assignment as described. The most interesting part was developing with RMI in mind, it was the first time I had combined such technology with normal threaded code and threads. The most challenging part was the gradual evolution of the code, requiring that I rigorously test the concurrent programming but also to explore design options that would best.

The project is generally laid out in three parts. There is a base, a client and a server project. All three need to be loaded into eclipse. The base project holds common code such as the DRS interface and several other classes that are used both by the client and server during communication. The server naturally implements the central server repository to manage the server list and also the retail server implementation of the DRS interface. The client project holds test code that can be executed to stress the server. I'll go over these in more detail in the following sections.

# Challenges and Assumptions

I've made several assumptions in doing this assignment that I'll stipulate. Firstly, this assignment has been coded entirely for a localhost solution. This may be refactored at a later date when the project nears. Each server additionally has a unique name that is the single uppercase character that begins a customerID. In order to manage the ports well I defined a starting port in the UDPMessage class, this is the port the central server uses and can be shared with clients if needed. All other ports for different servers can be determined by simply adding the character value for the server's name to the starting port. Thus a server's port will always be startingPort + storeCharacter.

In attempting to maximize concurrency I've tried to maximize the immutable nature and minimize shared access to data. This was especially apparent when coding the RetailStore (RS class). In order to maximize the concurrency, the inventory of all items was synchronized only on methods that modified the value of the stock (buy and return) but not on getStock. This was an acceptable trade off in concurrency since we definitely do not want two modifications at the same time but if old information gets to a user we can simply defend against it. At worst, the client will have to get a failed notification when stock was les s than advertised, something that happens at stores all the time.

Each retail server also maintains a map of customerID with open Logs. These logs are not synchronized since there is no need to guarantee the order of writing. They are time stamped however and each file is made per server, per customer.

One of the more interesting challenges was coding a separate UDP thread to be able to modify or retrieve information on the state of the retail server. To this end, I pass a reference to the RetailStore handling the RMI requests to the UDP thread. This thread then takes the actions of registering the new RetailStore to the central server and getting a list of current stores. This list stays up to date by the central server always repeating back to all current servers in the pool the full list. This is a sort of publish-subscribe model that is centrally managed. This means that any server can contact any other at any given moment with only a small window between registration and update. Importantly, the RSUDP thread must be able to modify the server list stored in the RS in order to update the list on new joins. This is done by maintaining the list of servers in a Set so that insertion of old servers never results in more than one entry. The set however must be accessed via a getServers method in the RS that makes an existing copy of the servers. Otherwise, the list may change if accessed again during UDP communication. In this sense, I've once again favored immutable data with a slight possibility of stale information over risk.

More importantly, the UDP thread had to be able to receive requests for stock information from others and send back. To do this, I've allowed the thread to access the current state from the retail store's inventory management. This is safe and unsynchronized since I have no worry about providing slightly inaccurate information to a client.

Another interesting challenge was the RS implementation of buyItems. In order to allow the full pool of items at all stores of a given itemID to be bought, I've used a recursive RMI call that will call the buyItems method at other stores if it cannot successfully meet a client request. This solution works well because it allows me to collapse back in failure if I reach the end of the server list or else I can buy from all stores before the end to make the purchase succeed.

# DRS-Base

This project contains the core of the interface specified between the client and the server, source links in both of the other projects read this code and use it in the communication. I'll explain the purpose of the classes in turn. Most of them are completely unrelated to each other so I haven't drawn any class diagrams to complement this explanation. Note, I've reused a modified PolicyFileLocator and RMIStarter from this tutorial: http://code.nomad-labs.com/2010/03/26/an-improved-rmi-tutorial-with-eclipse/. These code snippets aren't major and were released as a public tutorial so I see no problem in using them.

## DRServer

This class defines the main interface for the RMI methods and contains string names to be used in the construction of the specific registry name bound to each server store. This interface is critical to the overall functioning of the RMI methods. The client uses it to hold the stub retrieved from the registry. On the server side, the RS (RetailStore) class implements all the required methods.

## PolicyFileLocator

Simple class to handle finding the policy file that defines the permissions for the RMI operations. Simply creates a temporary copy of the original file and returns it to the caller. Used by the RMIStarter class later.

## RMIStarter

Base class for the server that handles several operations needed to run properly. It set's a few properties and makes available a class to the system based on the descendant class. Merely for convenience while developing and testing, could be folded into the current ServerStarter class.

## StockAvailable

This class is used by the CheckStock RMI method to return a map of the stores and their stock to the client. It serves no other purpose than to encapsulate this result and be used in the interface declaration of DRServer.java.

## TransactionResult

Similar to StockAvailable this class exists only to return a result to the client. In this case, the class encapsulates a String and a boolean. The string is a message the user may read to see the result of the transaction, the boolean value will be true only when the result suceeded.

## UDPMessage

An important class, UDPMessage contains the marshalling logic for all the UDP messages that are used on the server side. It has constants that define the starting port for the range of ports used by the servers. It has the ability to create a message from a series of arguments and a command and then flatten it into a byte array. This array can then be transmitted to a waiting UDP socket. The received byte array is then parsed by the UDPMessage class on the receiving side and the recipient may query the command/arguments sent to do work and then send a response.

# DRS-Client

There isn't much to talk about in this section. The client's simply represent a bunch of test classes that inherit from a common base. Each of the main client implementations tests certain test cases as listed in that section. Simply make sure that the server and registry are running and then execute a test case.

# DRS-Server

The core of the server lies here, I've described most of the major parts in the challenges above. I'll go over the classes quickly:

## Log

This class serves as the server logger. It is simple and is designed to be opened by each server whenever a client makes a request to a particular server. The logs of the server will be stored in a directory with the server's name. Inside that directory, each file will bear the customer's ID. All returns and purchases will be logged into this folder including timestamp for traceability purposes. There is no synchronization because it isn't needed really.

## Item

This class represent an item in a store's inventory, it has methods to get the current stock, return a quantity of the item or purchase some of the item. Only the buy and return methods are protected by synchronization since it would be unwise to allow multiple accesses to the same item. These items are managed in a Map contained in the RS where the key is the itemID known to be unique.

## RS

RS implements the DRServer interface, that means that it has methods for buyItems, returnItems and checkStock. The RS maintains the inventory, that is a map of itemIDs to items, a list of logs that is a mapping of customerIDs to log and a list of all servers currently in the pool in a set. These are the critical data structures in the project and as described before the amount of synchronization and data modification has been minimized to reduce possible bugs.

## RSUDP

RSUDP is a thread logically attached to each RS. It acts as a communication point to send and receive UDP messages to other servers. It can both register the RS with the CentralServer and receive a new list of servers to add to the RS. It can also get a request for stock information from another server and send a response directly back. These are the primary functions. The access to getStock is not synchronized because as previously stated there is only minor concern if a user gets stale data. The adding of servers is synchronized to prevent modification while a call to the getServers method is being made. This is done to protect the checkStock method in RS by having it use a copy of the server list while RSUDP is free to modify it at any time it isn't being read.

## CentralServer

The central server acts simply as a relay point for all the DRS servers. Each server coming online must send a join request to this server on a port identified in the UDPMessage class. This server will then relay back to all others the complete list of all servers in the current pool. This method means that all servers will always have the names of all other servers at almost any given moment and won't have to contact CentralServer for every UDP request.

## ServerStarter

This class acts as a wrapper for starting all of the servers in one action. It first creates an instance of the central server and then binds the correct RMI strings to their relevant DRSImplementations. It has no other major functions, see the Client project for test cases to run once started.

# Test Cases

I'll provide a list of test cases that I've written and passed. All test code to stimulate the server can be found in the drs.client package.

**Test Case 1: CheckStock(int itemID), item exists.**

The client should be able to execute a call to check stock on its closest server. That server will then be responsible to retrieve the current stock values of all other servers concurrently via UDP. The original server will then return the stock of the item. Since the item exists in at least one store, each stores particular stock is returned in a map relating the store name to the store stock of the given item.

See ClientCheckStock.java.

**Test Case 2: CheckStock(int itemID), item doesn't exist.**

The client should be able to execute a call to check stock on its closest server. That server will then be responsible to retrieve the current stock values of all other servers concurrently via UDP. The original server will then return the stock of the item. If the item does not exist, the store in question will return a stock of zero to the requesting server. These numbers will be returned to a client to evaluate.

See ClientCheckStock.java.

**Test Case 3: ReturnItems(customerID, itemID, numItems)**

The client will be able to determine the closest server to it and then make an RMI invocation to that server. There will be no failure state, it is assumed that any store can stock any amount of a returned item even if it is not presently in stock. If it is present, access the given item and increment stock. If not, create the item and increment the stock.

See ClientReturnItems.java.

**Test Case 4: BuyItems(customerID, itemID, numItems), first store has enough stock.**

The client will be able to determine the closest server to it and then make an RMI invocation to that server to purchase a set number of items of a particular item. The server should be able to check all other stores in order to fulfill the order if needed. In this case, the closest store has enough stock and simply returns a successful result to the client.

See ClientBuyItems.java.

**Test Case 5: BuyItems(customerID, itemID, numItems), no store has enough stock.**

The client will be able to determine the closest server to it and then make an RMI invocation to that server to purchase a set number of items of a particular item. The server should be able to check all other stores in order to fulfill the order if needed. In this case, the closest store doesn't have enough stock and will begin a recursive RMI invocation by first selecting the next server in the pool and invoking it's buy Items RMI method with the amount required, minus any the first store has in stock. This recursive behaviour will continue until it succeeds in this case returning a positive result to the customer.

See ClientBuyItems.java.

**Test Case 6: BuyItems(customerID, itemID, numItems), no store has enough stock.**

The client will be able to determine the closest server to it and then make an RMI invocation to that server to purchase a set number of items of a particular item. The server should be able to check all other stores in order to fulfill the order if needed. In this case, the closest store doesn't have enough stock and will begin a recursive RMI invocation by first selecting the next server in the pool and invoking it's buy Items RMI method with the amount required, minus any the first store has in stock. This recursive behaviour will continue until it fails at the last store in the list returning a negative result to the customer.

See ClientBuyItems.java.

All of the above transactions are logged and can be viewed in their respective logging folders. Also, some additional test cases with combinations of these may be built and demonstrated to show the proper concurrency is supported. They would simply be combinations of these transactions occurring simultaneously.