# Design Document 423 Assignment 2

# CORBA Distributed Retail Server

Jeremy Pallats

ID: 9619550

October 30th, 2012

# Introduction

The goal of this assignment was to take our existing RMI concurrent project and convert it to run on the CORBA Java IDL implementation. This required a moderate amount of code change, though many of the base classes were largely unmodified and could be reused. Mainly the RetailStore implementation was modified in order to use the new CORBA invocation system. Server objects are registered into the naming service and then requested as needed by clients and servers.

Additionally, a new method called ExchangeItems was required that allowed a user to return some items he had bought from the store and then immediately buy other items from the store. This method proved to be trivial to implement in comparison to the move to CORBA, it relies on existing code to checkStock and buyItems (the latter of which is atomic). The only substantive change in code as creating a LogParser that would allow a server to check it's permanent records and verify how much outstanding stock of an item a user had bought. Importantly, this will deduct any items he previously returned to the store via return items.

# Implementation and Assumptions

Important note, see README in code before attempting to execute.

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 RetailStoreServerImpl 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 less 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 remote 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. Due to this, 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.

The UDP thread also implements the ability to process requests for stock of a given itemID and also to check its own log and return the number of sold items to a user for the Exchange operation. Neither of these operations need be synchronized since we don't modify any data.

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. This recursive solution is atomic due to this collapsing nature and works well for ExchangeItems as well where atomicity is necessary.

# corba.base

This package contains the core of the interface specified between the client and the server, in the interface.idl file. Most of the other classes are simply utility functions that could be used by either client or server and so were put in a base package.

## DRServer

This class used to define the main java interface for the RMI operations. Now it simply acts as a placeholder for final strings that identify the methods required to access.

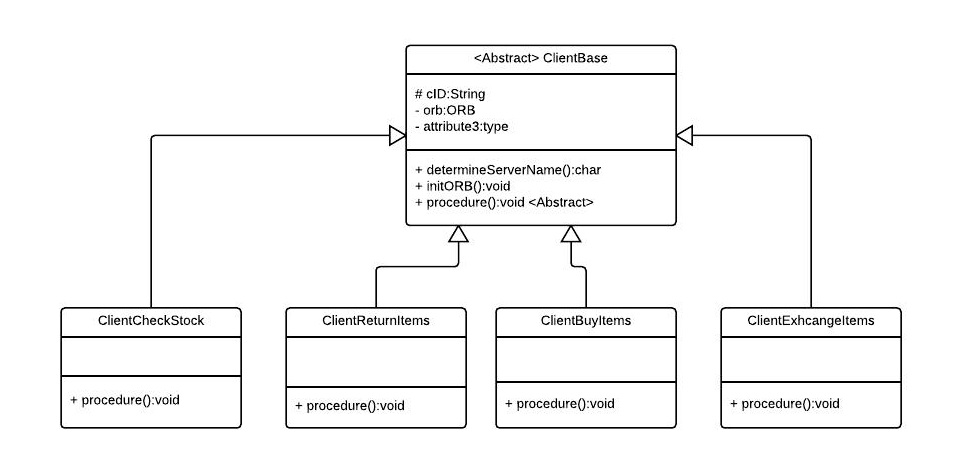
## StockAvailable

This class is used by the CheckStock method aggregate the map of stores to their available stock for a given itemID. Due to the changes for CORBA, this class is no longer returned to the client. Instead I've added a method to return an array of Stock\_s objects that are part of the definition.

## 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 back.

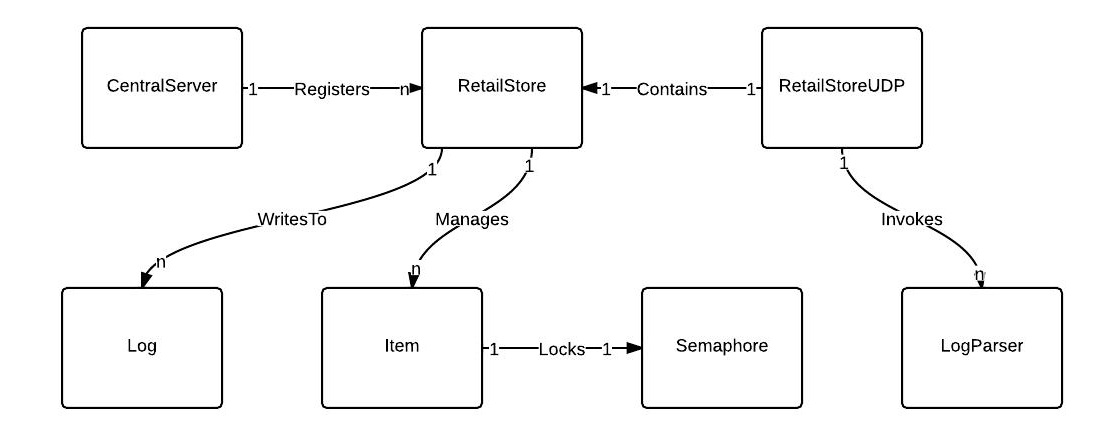
# corba.client

 All code in this section exists to verify the test cases listed at the end of this document. The ClientBase class forms an Abstract parent to all others and handles the ORB initialization based on the customerID and method required. The class has an abstract procedure method that descendants can override in order to implement the class. I've created a simple Class Diagram to show their relationship below.

# corba.server

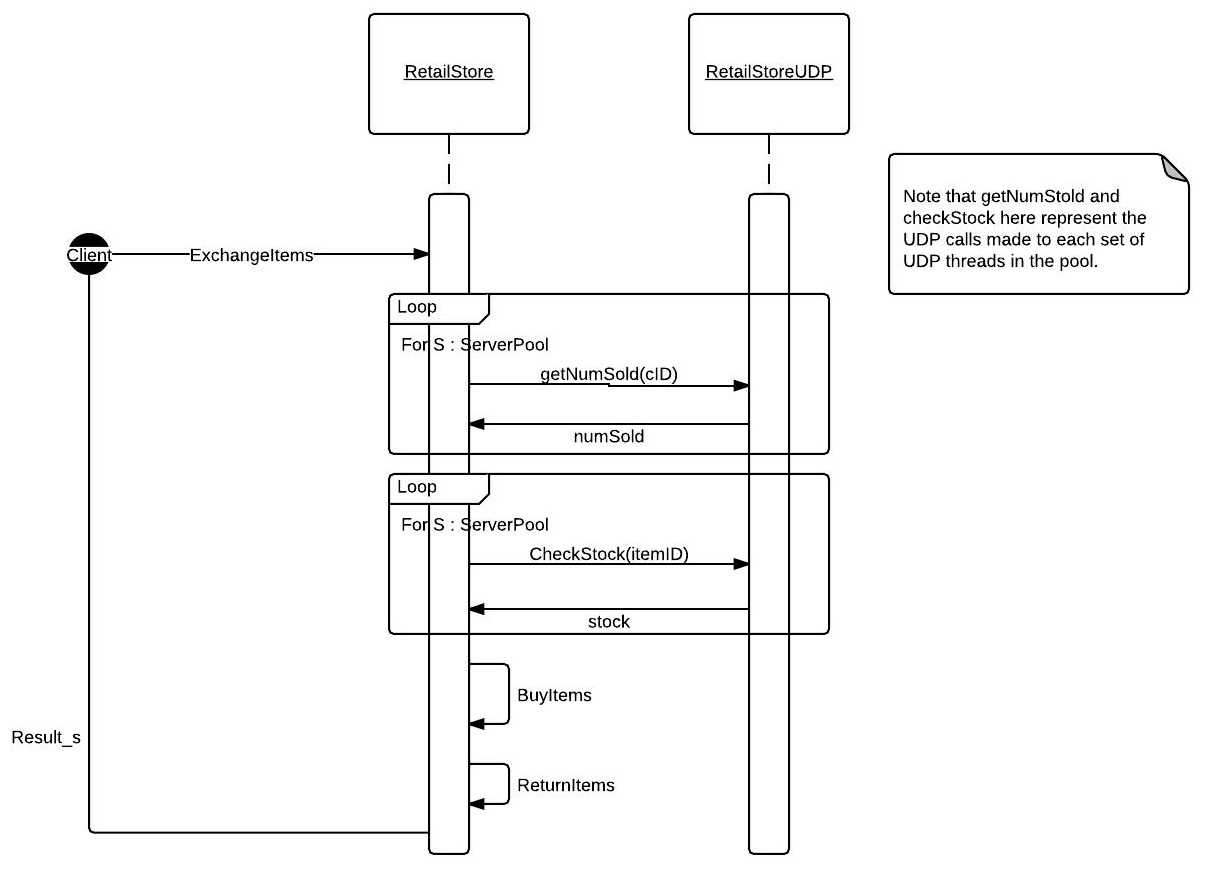
Figure : Client Class Diagram

The core of the server lies here, I've described most of the major parts in the implementation section above. I'll describe the core layout in this section and go over important changes from the first assignment, including the new ExchangeItems calls. The ER diagram below gives an overview of how the entities relate to one another.



## Server ExchangeItems Sequence

The below diagram summarizes the sequence of calls needed to fulfill the ExchangeItems request. Note that within the loop, the messages are all UDP packages and not actual methods. They have simply been named to represent their idea.



## 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, exchanges 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 modifications concurrently. These items are managed in a Map contained in the RS where the key is the itemID known to be unique.

## RetailStore

RetailStore implements the CORBA methods defined in the IDL interface, that means that it has methods for buyItems, returnItems, exchangeItems and checkStock. The store 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.

## RetailStoreUDP

The UDP thread is logically attached to each store. It acts as a communication point to send and receive UDP messages to other servers. It can both register the store with the CentralServer and receive a new list of servers to the store's server list. It also supports several UDP requests such as requests for stock of an itemID and also a request for the number of items sold to a customer. 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 not synchronized since a copy of existing servers is always made when they are needed.

## 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.

## CorbaStarter

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 corba.client package.

**Test Case 1:** CheckStock(int itemID)

**Class:** corba.client.ClientCheckStock.java

**Purpose:** Verify CheckStock functions.

- If itemID exists in no stores, expect every stock value to be zero.

- If itemID exists in some stores, expect at least 1 non-zero stock value.

**Test Data:** itemID, an integer that is the item to check in inventory.

**Steps:**

1. Client determines closest server.
2. Client sends a checkStock request for an itemID that exists.
3. Server fills in its own inventory stock for this itemID.
   1. If itemID in inventory, use stock number.
   2. If itemID does not exist, fill in 0.
4. Server contacted initiates a concurrent UDP request to all other UDP threads to get their inventory for the itemID, follows the same logic as above.
5. Each RetailStoreUDP thread replies with a UDP packet that contains their own stock of itemID.
6. Original server combines these values into a single Stock\_s array and returns to client.

**Test Case2:** ReturnItems(customerID, itemID, numItems)

**Class:** corba.client.ClientReturnItems.java

**Purpose:** Verify returnItems works. Succeeds under all conditions, never performs any checks.

**Test Data:** itemID, an integer that is the item to return. customerID is a customer that may or may not exist. numItems is any integer > 0.

**Steps:**

1. Client determines closest server.
2. Client sends a returnItems request with the required arguments.
3. Server makes no checks, accepts the return, updates inventory and then logs the transaction.
4. Initial server returns a Result\_s object that informs client of success.

**Test Case 3:** BuyItems(customerID, itemID, numItems), first store can satisfy request.

**Class:** corba.client.ClientBuyItems.java

**Purpose:** Verify that function suceeds unless:

- A semaphore on the required item was taken, client should wait and then request again.

- All servers do not have enough stock to complete.

**Test Data:** itemID, an integer that is the item to buy. customerID is a customer that may or may not exist. numItems is any integer > 0.

**Steps:**

1. Client determines closest server.
2. Client sends a buyItems request with the required arguments.
3. Server attempts to get semaphore lock on item requested.
   1. Failure to acquire the semaphore results in immediate failure.
4. Server checks local stock.
   1. If it has enough stock, complete transaction at this store and return success to caller.
5. Server calculates the number of itemID required at other stores.
6. Server determines next server to contact and makes a recursive call to buyItems with number of items required elsewhere.
7. Recursion continues going to each store and performing steps 3-6 until...
   1. A server fails to acquire the semaphore, collapse back with failure to be returned to client.
   2. Server contacted has enough stock locally, complete purchase and then return success.
   3. Base case reached where next server to contact is originator, fail and inform client.
8. At this point, call must collapse back in success, each store completes transaction and releases the locked semaphore on the item. The purchases are all logged locally.
9. Client is informed of ultimate success of the call.

**Test Case 4:** ExchangeItems(customerID, boughtItemID, boughtNumItems, desiredItemID, desiredNumItems).

**Class:** corba.client.ClientExchangeItems.java

**Purpose:** Verify ExchangeItems call under successful conditions. Additionally, verify failure when:  
 - Customer didn't purchase boughtItems from DRS stores.

- Stores do not have enough combined stock.

- BuyItems call failed, causing an atomic fail.

**Test Data:** customerID is a customer that must exist prior to request. desiredItemID must currently be in system. boughtNumItems and desiredNumItems > 0.

**Steps:**

1. Client determines closest server.
2. Client sends an exchangeItems request with the required arguments.
3. Server initiates a request to get the number of boughtItemID sold to customerID by UDP.
4. RetailStoreUDP threads concurrently parse log for customerID and return stock sold - stock returned for boughtItemID.
5. If customer did not buy at least boughtItemID previously, fail.
6. Server initiates a checkStock request for desiredItemID.
7. RetailStoreUDP threads concurrently check store stock for desiredItemID and return it in reply packet to origin.
8. If store stock for desiredItemID is less than desiredNumItems, fail and inform user.
9. Original server now attempts to buy the desiredNumItems of desiredItemID using buyItems.
   1. If transaction fails, instantly fail and inform user to try again at later date.
   2. If it succeeds, log a return of boughtItemID at this store and complete transaction with succeeds.
10. If client receives failure, may wish to attempt to retry with exponential back off to see if it was a temporary locking problem.

All of the above transactions are logged and can be viewed in their respective logging folders (except checkStock). 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.