University of Glasgow | School of Computing Science

# Distributed Algorithms & Systems 4
# Auction System Report

Andrei-Mihai Nicolae (2147392)

November 24, 2016

**Overview**

The task was to design, implement and test an auctioning system using Java RMI. It should allow concurrent access from multiple clients to a server that can handle bidding, displaying a particular auction based on its ID, display all auctions that are currently available and many other features that will be discussed in the report.The task has been successfully completed, having been tested both locally and remotely (i.e. both server and client on the same machine, server on one machine and client on another). All requirements specified in the specs sheet have been implemented correctly and work in a rather efficient manner.

## 1 Design

The project started with defining the basic functionality of the system: the server should allow concurrent access from multiple clients and deal with requests accordingly. Therefore, I have devised my work into 3 main parts: defining the interfaces that the client uses to communicate with the server, creating the client and creating the server. There are also some extra classes I implemented such as a date formatting utility or a system evaluator, but I will not go into in-depth detail over those.

### 1.1 `AuctionManager`

I have created 2 interfaces with corresponding implementations: one is the AuctionManager class (its implementation is the AuctionManagerImpl class), which handles all requests coming from the server and updates auctions, places a new bid on an auction specified by the user, allows saving and restoring state from permanent storage (also used for bootstrapping the system if there is any state saved), displays more detailed information regarding a specific auction or summarized info for all auctions created, as well as a method that is used by the client to detect if the server down and act accordingly.

The Auction Manager is also able to delete closed auctions (regardless if the starting price was met by any bidder) after a configurable unit time defined in the class. Moreover, it can notify users of various interactions that are happening in the system: if the user has won an auction (also the other users are notified that someone won the auction with a specific amount of money).

### 1.2 `AuctionParticipant`

The AuctionParticipant interface (with its implementation AuctionParticipantImpl) is defining a user of the system. When the client starts, it will also create a new user with the name parsed by a scanner. Then, I decided to use a rather simple interface for it, giving it only 3 methods: `getName()`, `getId()` and `notify(String)`. The last method is the most important as it is used by the server to notify the user when specific events occur (e.g. the user won an auction, his/her item was sold for x pounds).

### 1.3 `AuctionClient`

I tried to make the client as user-friendly as possible, implementing features such as: asking the user for name, the ability to always type help to see the list of available commands if they were forgotten, as well as letting him now of a variety of things through the notification mechanism I mentioned above.

It will go through and loop until user has either typed in quit or the server has been shut down/quit unexpectedly. I have also implemented easy to understand errors informing the user of any error such as typing a wrong command, placing a bid smaller than the item's current value, displaying an auction that hasn't been created yet, as well as inputting the closing time for the auction in an invalid format.

I have created also created a network failure detector which is a single thread running continuously that will call a dummy method on the server and, if the timeout reaches 5 seconds, it will assume that the server is down and it will stop the client automatically.

Coming back to the core functionality of the client, I have made it look-up the address and port specified by the user as arguments when running the class. It will try to see if there is an Auction Manager bound to that address (using `Naming.lookup()` method) and, if so, it will start waiting for commands from the user. As stated before, the user can choose to do one of the following: bid, create auction, display specific auction, display all auctions, save state, quit the client as well as display all possible commands via help.

### 1.4 `AuctionServer`

The server class is relatively simple. Firstly, it will create an RMI Registry via `LocateRegistry.createRegistry(portNumber)` method call. This will make that registry available on the port specified inside. Then, it will create a new `AuctionManager` object that will handle all the processing and it will bind the object to address `rmi://localhost:1099/AuctionServerService`.

Then, it will wait for any incoming connections from clients locally or remotely. I have also set the `java.rmi.server.hostname` environment variable for remote access because, when I tested it on different lab machines, I had errors that the clients could not connect to the specified address (even though it was typed in correctly).

For both the server and client, in order to function properly, I have added client and server policies files that grants them all security permissions available.

Furthermore, as I mentioned before, when a user types in `save`, the Manager will save the state in a bin file in the resources folder. When the server loads, it will check if there is any bin file on permanent storage and, if so, it will allow bootstrapping the server with the saved auctions.

### 1.5 `Auction`

This is the class for defining a simple auction. The Manager will create auctions, place bids and so on calling the methods defined in this class.

The Auction contains a Timer that will start once it is created. First, it needs to have the closing date after the current time. If this is the case, it will start a `TimerTask` object that will run until the closing date and time.

Also, the class contains all the relevant fields to maintain consistent and useful information : it has a Map to store the bidders, the highest bidder at any time, an id that is guaranteed to be unique (used `synchronization` mechanism in the Manager to guarantee that even if we have multiple clients, each Auction will have a unique ID), as well as the item's title and current value. Furthermore, I made it implement the `Serializable` interface as we will pass these objects around.

### 1.6 User Interface

The user interface is very rudimentary. I am taking all the user's input commands via a `Scanner` object and displaying on `System.out` all performed actions. As I previously mentioned, the Manager employs notifications to users informing them about relevant events such as the user has won an auction or his item was sold to someone.

### 1.7 Testing Suite

Even though the testing suite I have written is not so relevant when the server and clients are spread across the network, I wanted to test if the system meets the requirements on local machine. I created tests to see if bootstrapping the server and bidding/displaying information etc. work. I also wrote a smoke test to see if building from scratch a server and a client is stable.

### 1.8 Build Automation

I have ensured that anyone trying to run the system will have a pleasant experience by creating a `Makefile` and scripts for running all components: client, server and tests (or specific ones if one does not want to run all of them).

## Possible Extensions

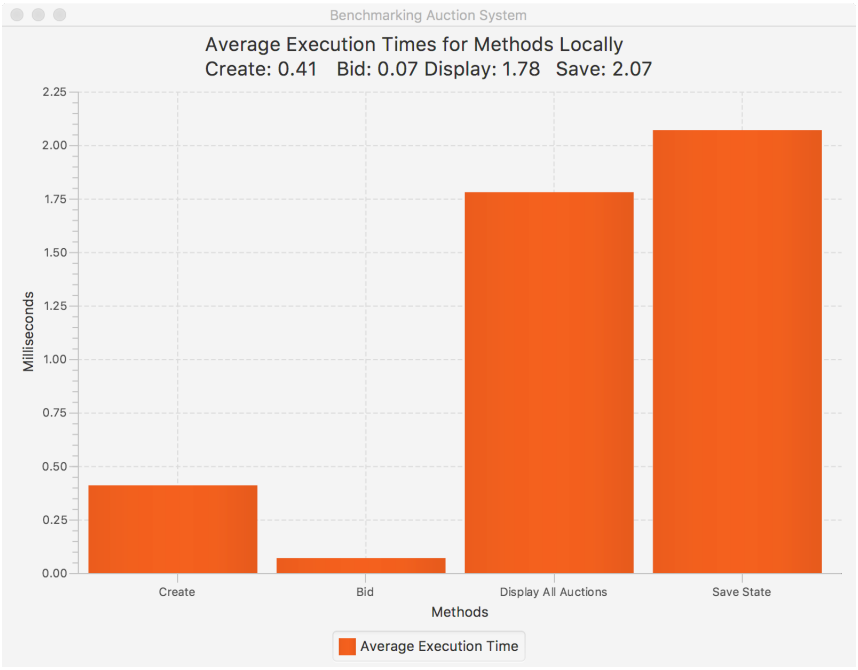
Possible improvements to my implementation could be:

- Add multiple servers so that if one of them fails, the whole system does not crash. This could be done by balancing workload and if a failure is detected, one of the other servers that is running can do more processing. This would require a major change, but it is mandatory if the auctioning system would grow in scale.
- Definitely create better User Interface and User Experience. Maybe some JavaFX can be added to the system and make a pleasant Graphical User Interface. One example of benefit if there were a GUI could be that typing commands would be replaced by clicking buttons, which is a far better interaction with the system.
- The system should log at regular intervals and these should be stored somewhere on permanent storage. One advantage to this approach would be that debugging failures becomes easier.
- Create a database of users when they log in to the system so that they can retrieve their own history of auctions (i.e. which ones they have placed a bid on, what items did they sell, what items did they buy etc.). This would require a password associated to the account (email address also), but this should be manageable.
- Maybe create a domain name for our server's IP address so that it will be easier to connect to.

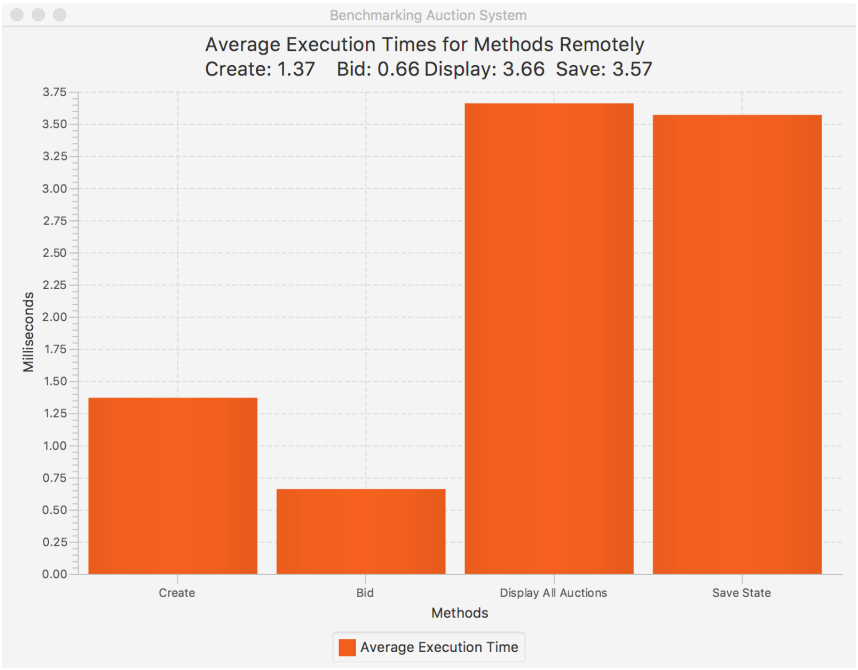
## Performance

As I've mentioned, I have tested the system both locally and remotely. They provided quite interesting and valuable input to my testing and performance analysis.

I have deployed a simple testing interface that uses `JavaFX` to build a basic bar chart. There, I have outputted average execution times for Manager methods (create, bid, display auctions and save state) for a certain number of runs. Everything can be configurable by the user (host name, port number and number of executions). The class that implements this functionality is called `EvaluatePerformance`.




As we can see in Figure 1, the numbers are rather small for this scenario. Everything performs fast, apart from the last two. However, this slow execution is normal due to both writing the state to disk and outputting an increasing list of auctions with all their details.




However, in the second image you can notify the rather significant increase in execution times. The methods that were called took a much longer time due to machines being spread across the network. It is true that the load on each machine counted towards these results, but after multiple tests the execution times were always much higher than running everything on local.

## Testing the System

In order to test the system, please refer to the following instructions:

$ `tar -xvzf 2147392.tar.gz`

$ `cd 2147392`

$ `make`

$ `chmod +x *.sh`

$ `./run-server.sh`

$ `./run-client.sh` (in another Terminal tab)

$ `./run-all-tests.sh` (in another Terminal tab)

$ `make clean`