Distributed Systems and Algorithms Assessed Exercise

Name: Petre Stegaroiu
ID: 2082041S

How to run the system

1. Unzip 2082041s.zip to the desired location. I will use “C:\DAS” as location for reference
2. Open 3 terminals and go into “C:\DAS”
3. Compile all files by typing “javac *.java”
4. Run server on a terminal by typing “java Server”
5. Notice that the server bootstrapped 10 auctions.
6. On the other terminals run Client by typing “java Client”
7. Notice that the clients are given instructions on how to create an auction, bid , see available auctions and store auctions.
8. Test those commands and check that server and client respond accordingly
9. Test the fault tolerance of the system by disconnecting the server at any time or the client after he made a bid or created and auction.

Functional Requirements

1. Auction System should consist of an auctioning server and client programs
2. Client should be able to create auction items by specifying their name, minimum value, closing time in seconds.
3. Creation of item should return a unique id
4. Client should be able to bid on existing auctions
5. Client should be able to see all available auctions
6. At the end of auction, owner as well as bidders of the item should be notified of the result (winning id, winning bid, price not met)
7. Server should handle dynamically changing set of auctions
8. Server should deal with client requests
9. Server should maintain state of ongoing auctions

10. Auction information should be available to clients for a certain time period after auction closes
11. Server should be able to save auction state to permanent storage
12. Server should be able to save restore auction state from permanent storage which should be used to bootstrap initial auction items for testing
13. Auction System should have a failure detector implemented so that clients can test if the server is still responding
14. Auction System should deal with issues regarding concurrent client access
15. Auction System should provide a simple text-based interface for clients to operate with

Analysis of the System

My auctioning system consists of a client class(Client.java), a server class(Server.java) an auction system through which the client communicates with the server as well as a performance testing class(RMIPerformance.java).

Client

The client tries to look up the auction system remote object at a given address. If the server bound the auction system at the given address then the system increments its “ownerid” field by 1 and gives it to the client in order to uniquely identify him. The system also shows the client a set of instructions that inform him how to interact with the text based interface as well as how to quit.

The interface has been designed to be user friendly. It explains in detail how each command should be typed in and what to expect as a result. If the client makes any input mismatch exception then the system handles the exception by either explaining his mistake or reminding the client what commands he is allowed to use. The client is also provided with confirmation of the success of his command.

Server

The server class is very lightweight, pushing most of the implementation to the auction system class. The server binds the auction system remote object to a specified address and also runs the rmiregistry in order to avoid having to run it manually on the

command line. It also provides a wrapper method for the auction object to signal the system to remove itself from the list of available auctions once it has expired.

Once the server starts it tries to restore all auctions from a specific storage file(storage.ser) so that the client can bid on them. Restoring auctions is done by using the fields of the auctions from the storage in order to recreate them again. As a result, the server does not keep the state of the auction(bidders, highest bid, time remaining till it closes). This is not to be considered an issue because the main purpose of restoring auctions is for the server to bootstrap a list of auctions for the clients to bid on. The server automatically bootstraps 10 dummy auctions in case the storage file does not exist or there are auctions stored within it.

I have tested the restore functionality by starting the server without having a storage file and confirmed that the server bootstrapped 10 auctions by connecting with Client and asking to see all available auctions. Then, as a client, I created a new auction and asked the system to store all auctions. After reconnecting again to the server I noticed that the 10 bootstrapped auctions together with the one created by me have been restored.

Auction System

The Auction system interface allows the client to use its methods in order to create auctions, place bids , see available auctions, store all auctions on the server permanently and ping the server.

Creating auctions

The system generates a new unique auction id by incrementing its value by 1 each time the client creates an auction. This id together with the owner id and the parameters given by client(“createAuction <name> <minValue> <closeTime>”) are used to create an auction object. If the creation of the auction is carried out then the client is informed of the auctions id.

The system dynamically stores the list of auction items into a hashmap with keys being the auctions unique ids. The hashmap data structure was chosen so that the system can quickly access the auction based on auction id. The advantages of this choice can be clearly seen when clients bid on auctions using their unique ids since accessing an element within the hashMap takes O(1) compared to O(n) in the case of choosing a list.

To test this specification I connected as a client, created an auction and then asked the system to show me all available auctions. I noticed that the server sent back confirmation of the creation of the auction as well as the auction.

Bidding on auctions

The auction object stores the owner id, auction id, name, minimum value, closing time as well as a list of all bidders and the highest bid object. A bid object contains the client's owner id and the price he provided.

When the client places a bid on an auction by typing "bid <auctionID> <value>", the system first ensures that the auction exists, then it checks that it is still open and finally it makes sure that the price of the bid is higher than the minimum value of the auction. If any of those conditions is not met then the system informs the bidder of why the bid was unsuccessful i.e the auction id is wrong, the auction is closed or that he has to bid more than the minimum value. Otherwise, the auction updates its minimum value to the bidder's price as well as creating a new bid object which is used to update the highest bid field of the auction. The auction also updates its bidders list in case the client made his first successful bid.

I tested this functionality by bidding on a non-existing auction, a closed auction. In both cases, the server informed me of the reason behind the failure of the bid. This happens also if I try to bid on an auction with a price smaller than the minimum value. Finally, I did a valid bid and got confirmation of my command.

Auction closing time and expiry

In order to inform the owner as well as the bidders of the result of an auction, the server automatically registers clients with the auction object when they create the auction or when they place a successful bid. This is done through the register method located on the auctions interface. The auction keeps a list of all those registered clients in order to make a callback through each client's interface in order to tell them the result.

Once an auction is created, the auction is scheduled to close after closeTime (provided by client) seconds. At closing time the auction sets itself as closed by updating the "hasClosed" boolean flag, it callbacks all of the registered clients informing them of the result as well as scheduling itself to expire after a certain amount of time.

The expiry time is a hardcoded constant shared by all auctions, for testing purposes it has been set to 60 seconds. Once the auction expires, it sets itself as expired in order to allow flexibility in case of specification changes regarding expired items. It also asks the server to remove the auction from the list of auctions. As a result, neither the client nor the server have any more access to that auction object.

To test this specification, I created an auction and specified that it should close in 5 seconds then bidded on it and 5 seconds later the server returned back the message “Auction 10 has closed. Your id is : 0. Client with id 0 has won the auction for the price of 10£”. 60 seconds later I verified that the auction is not displayed anymore through show auctions command.

Show available auctions

The client can request the server to see all available items by typing “showAuctions”. The server goes through the list of available auctions and returns all the information back to the client as a string. This is done by appending the id, name, price and whether it closed or open information for each auction in the list to a stringbuilder and returning its string value. I choose this implementation rather than returning a list of auctions to the client in order to reduce the amount of data to be sent over the network.

Storing auctions permanently

The client can choose to store all auctions available on the server to a permanent storage file(storage.ser) by typing “storeAllAuctions”. The auction system takes this request and puts all its auction objects on the storage file. Once the transfer is complete the client is informed of the result of the storage i.e “Storage failed” or the list of auctions that have been stored. In the current implementation, each storage call of the client overrides the previous storage. This method has been implemented in order to allow the server to bootstrap the desired auctions.

Testing the performance of the system

RMIPerformance class tests the performance of the system by calling a certain method(createAuction, bid, showAuctions) a given amount of times in order to get the average call time(call/ms) and then repeating this process several times in order to get a list of average call times. It then counts how many times each time appears and creates a barchart showing all

this information, the overall average it took to make that call and how many calls can the system make per millisecond.

Fault detection

The client pings the server every second to make the server is still responding. If the ping call is met with an exception then the server is down and the client is informed of the situation as well as disconnected. This way the client immediately becomes aware of the crash of the server. I prefer the client probing the server rather than the server calling back to the clients periodically. This is because I believe it is more important to know when the server is down rather than being informed that it is still running. Another reason would be that the messages sent from the server would become spam for the client. I tested this specification by running the server, connecting the client and verifying that the client immediately gets disconnected once I close the server.

The system deals with the issue of the client disconnecting after having created an auction or having made a bid and before the server could callback to the client with the result. It does so by catching the remote exception generated by attempting to callback to disconnected client and printing on the server interface that this client is not connected anymore. In order to test this, I created an auction and told it to close in 5 seconds, then I closed the client and noticed that 5 seconds later the server is aware of the client being down.

Concurrent client access

The first concurrent issue that the system covers is assigning each client a unique id when they connect to the server. If 2 clients would connect at the exact time it might happen that the system's "ownerid" field gets incremented twice before being returned to any client and as such both clients would get the same id. In order to prevent this problem, I have prevented thread interference to the "ownerid" field by making it of type "AtomicLong".

I have used the same approach in order to solve the problem of concurrent creation of auction objects which could result in assigning the same id to different auctions. Hence, the auction id field is also of "AtomicLong" type.

The final concurrency issue that I dealt with was concurrent bidding. The problem is that the comparison between the bid's price and the auction's minimum value as well

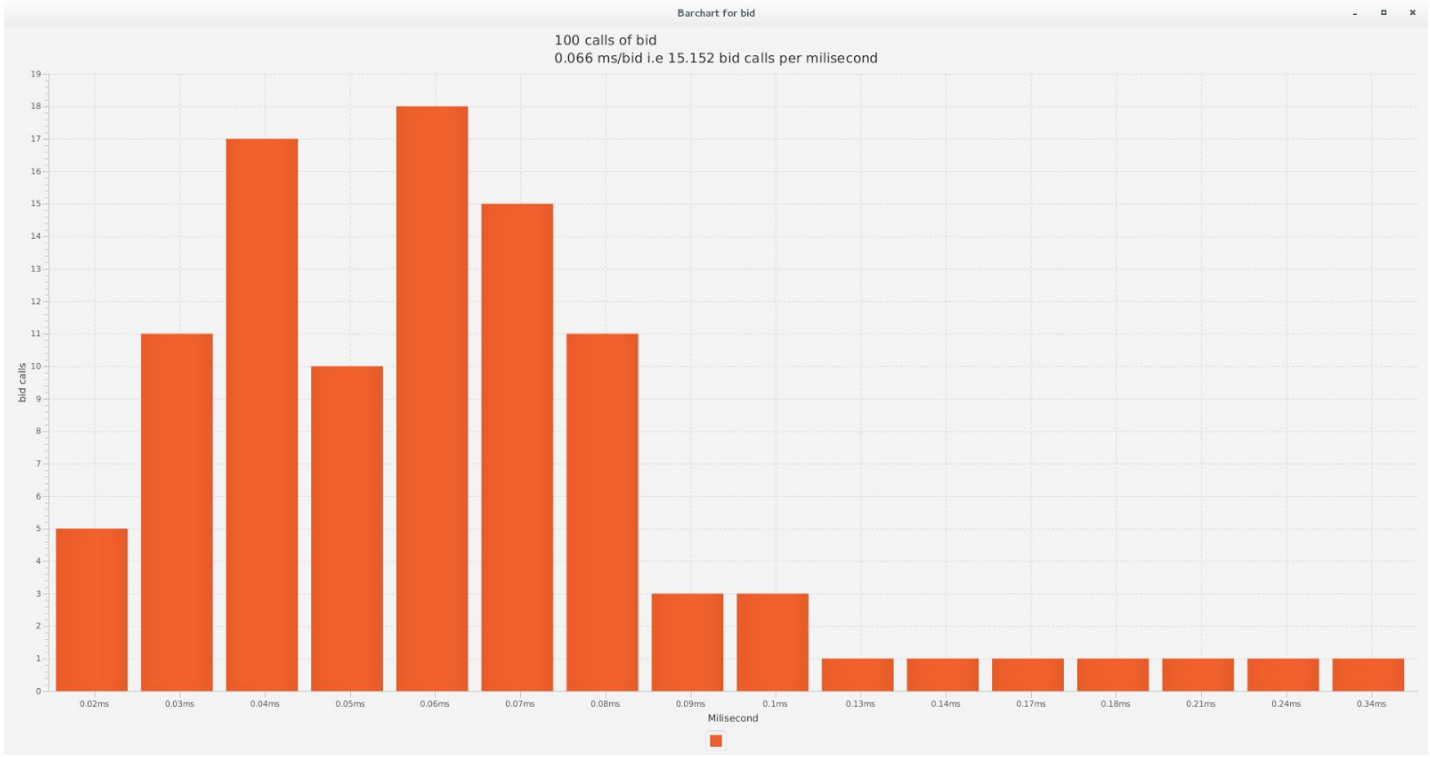
as updating the auction's highest bidder and minimum value must be done synchronously, otherwise a client might be allowed to make a bid on the auction although another client would have made a higher bid. This would result in faulty behaviour as the auction's minimum value and highest bid would get the wrong values. In order to prevent this behaviour, my system synchronises those aspects of the bidding method.

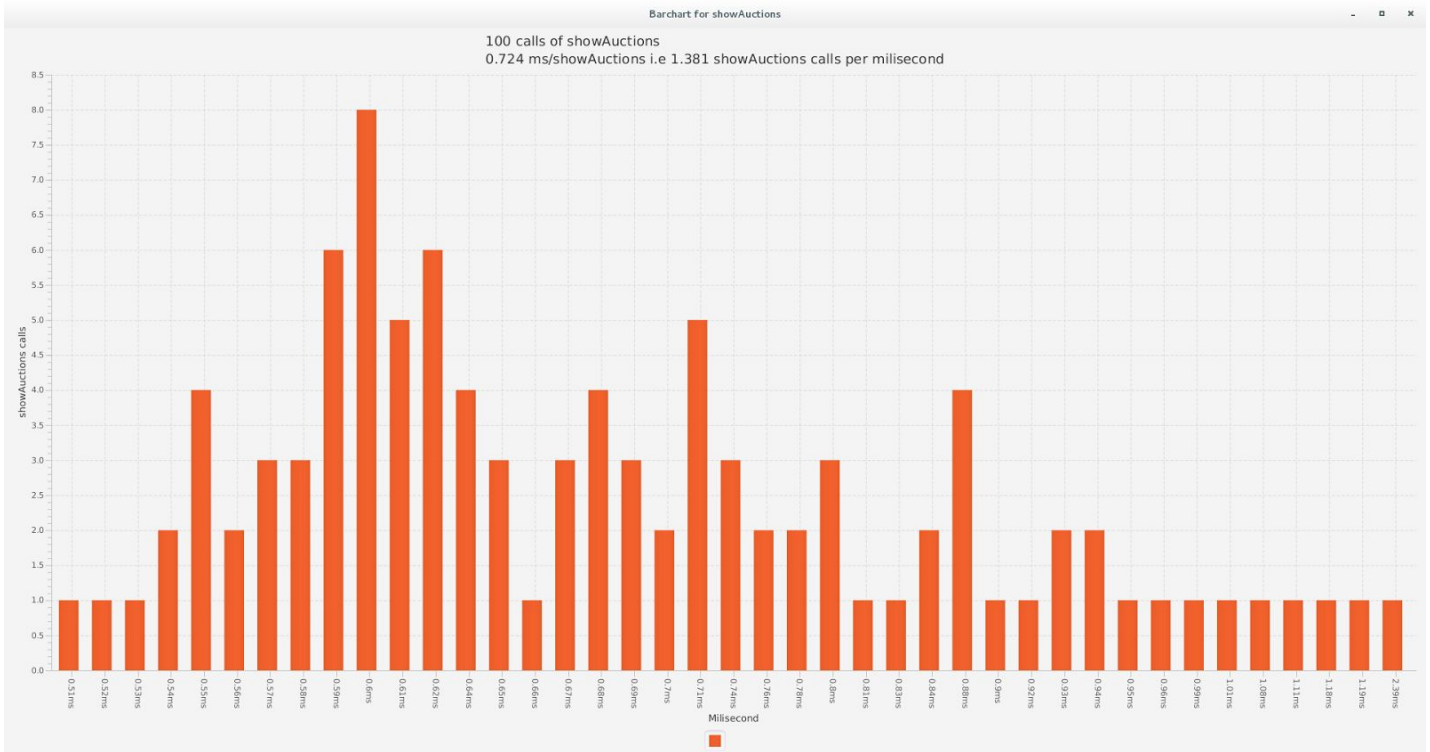
In order to test the against concurrency errors I ran the RMIPerformance class on 2 different terminals at the same time, once calling bid and the other time createAuction. I noticed that there have been no concurrency exceptions and after connecting with the client I checked that all the auctions were created with correct bid values.

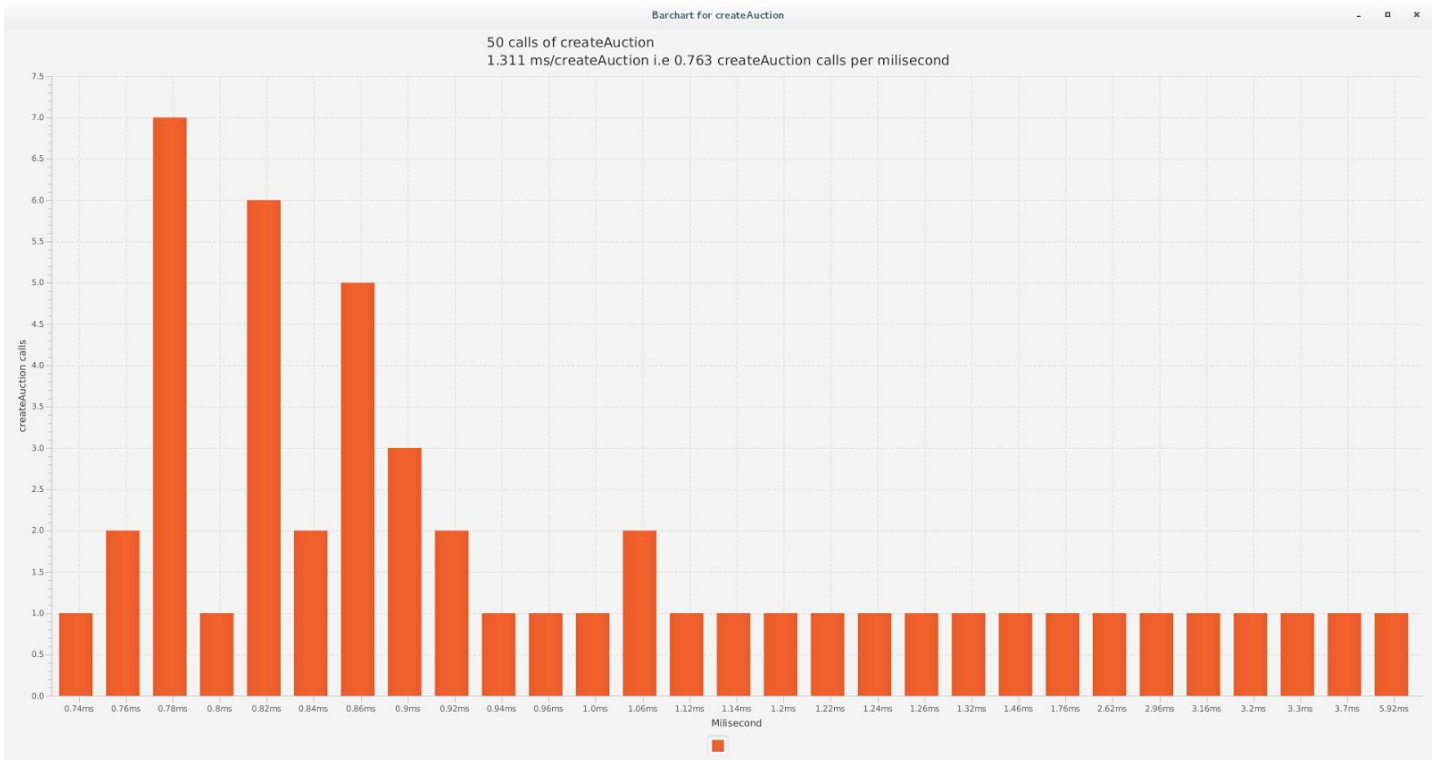
Performance evaluation

I ran RMIPerformance class using all 3 main client methods(createAuction, bid and showAuctions) both locally and remotely in order to find out how much time each invocation takes and to evaluate how much faster it is to use system locally. The barcharts show the average invocation times of running those 3 commands 100 times for bidding and showing auctions and only 50 times for creating auctions because it runs out of memory on lab machines if I try with 100. Showing auctions method is set to only display 1 auction, the time it takes for this command to run depends on how many available auctions are on the system. Below are shown the barcharts generated by the RMIPerformance class:

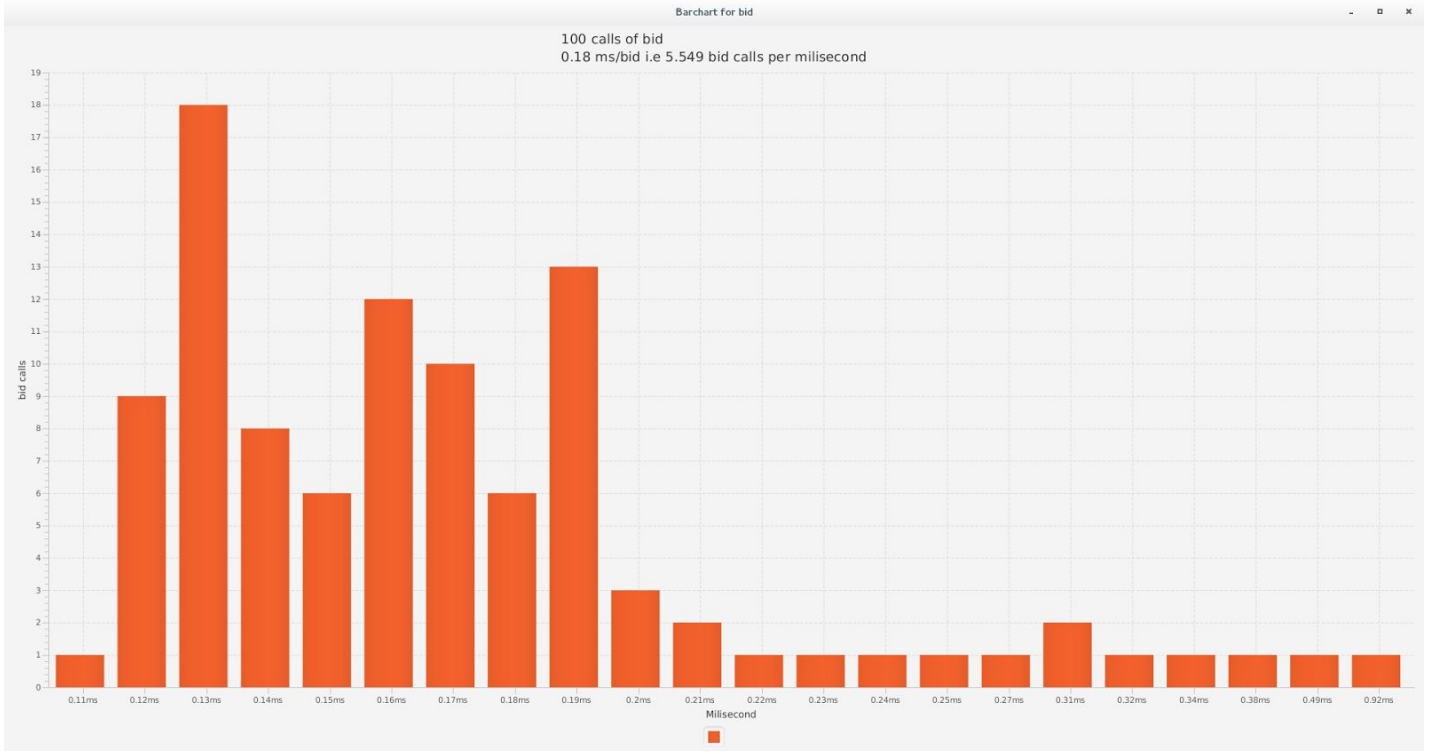
Invoked remotely

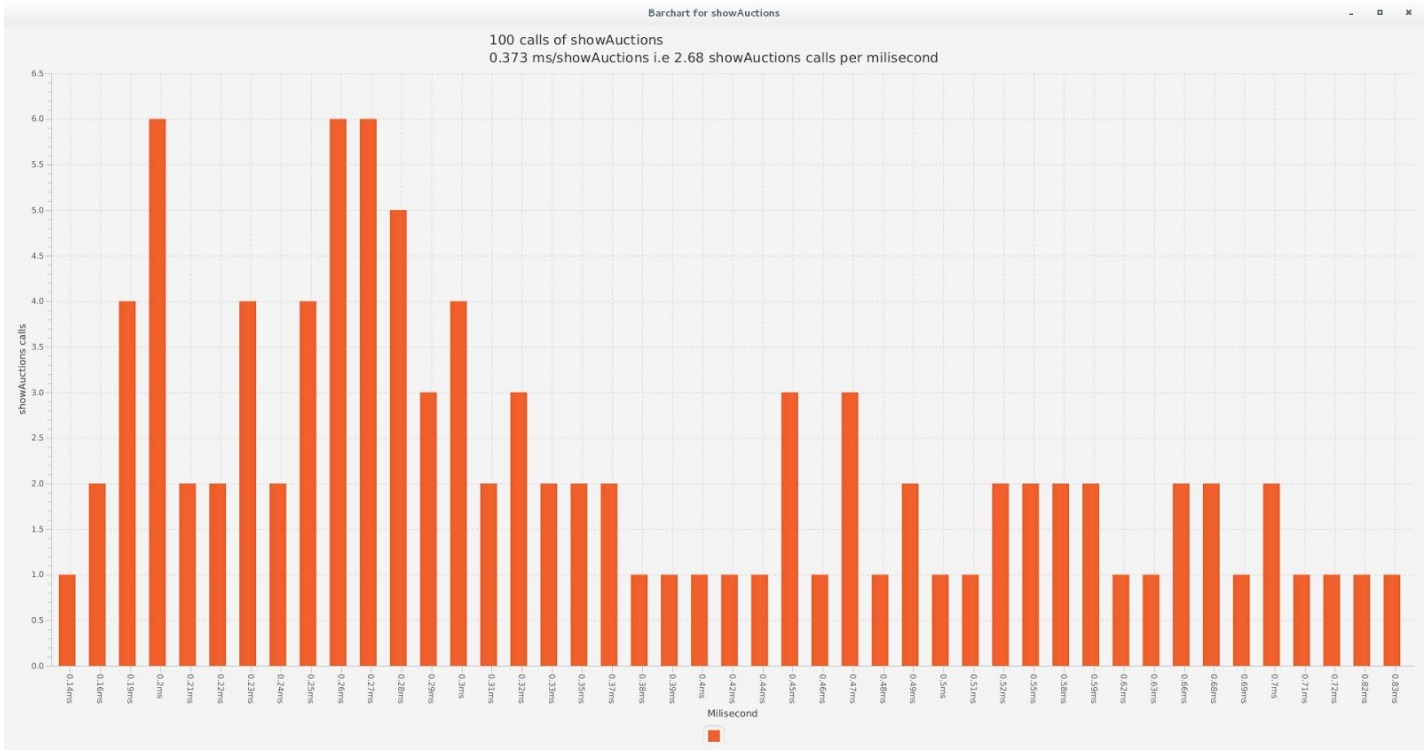


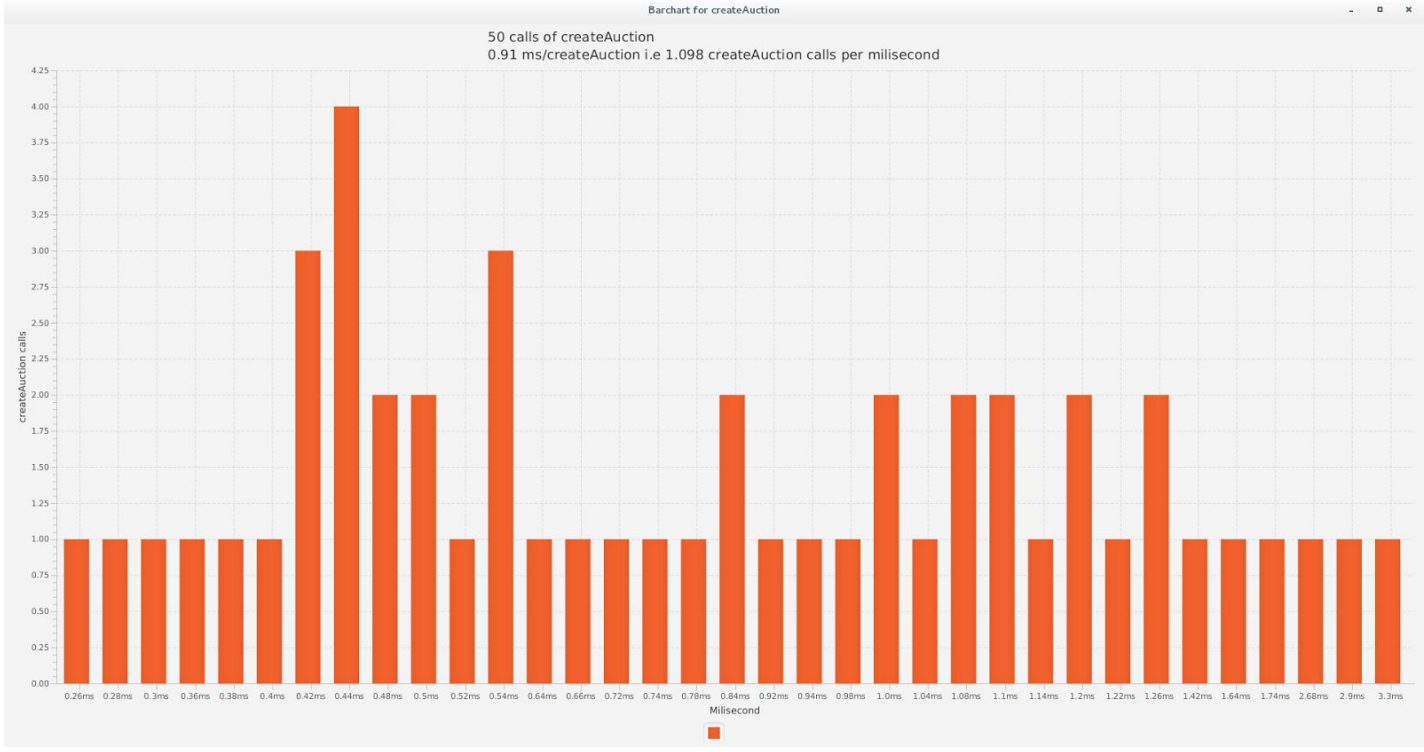




Invoked locally




Barchart for showAuctions
100 calls of showAuctions
0.373 ms/showAuctions i.e 2.68 showAuctions calls per milisecond
showAuctions calls
Milisecond


Barchart for createAuction
50 calls of createAuction
0.91 ms/createAuction i.e 1.098 createAuction calls per milisecond
createAuction calls
Milisecond

The bar charts above give us those results for average time per invocation:

Method	Remote call	Local call
bid:	0.066 ms/call	0.018 ms/call
showAuctions:	0.724 ms/call	0.373 ms/call
createAuction:	1.311 ms/call	0.910 ms/call

The results above show that the remote delay becomes less and less significant as the commands become more time consuming i.e bidding is 3.66 times faster remotely, showing the client 1 auction is twice faster while creating an auction is only 1.5 times faster.

The bar charts also show that the mean value of average invocation time is strongly influenced by extreme points. This can be clearly seen in the createAuction barchart invoked remotely where the overall average is 1.311 ms/call even though the majority of invocations take around 0.8 ms/call.

Potential extensions

An improvement for the system would be having more than a single server. Having multiple servers would improve the availability and fault tolerance of the system because splitting the load of the server into several smaller ones over a network would decrease the likelihood of out of memory exceptions. It also means that if a server crashes not all clients are disconnected. The performance of the system would also increase as the servers would respond faster. This extension can be done by assigning a server for different auction categories similar to what eBay provides. The client would connect to the server that has the auctions he is interested in.

Implementing a security manager would improve the fault tolerance of the system as it would allow the RMI to download code from the remote machine. In my current implementation the server and the client are within the same package and as such extending the system with security manager would not provide any benefit. However, in order to make the system better structured and to allow the client to run without requiring other classes I would split my implementation into client, server and auction system packages. Hence, this will require a security manager. This extension would be done within the server class by checking if there is any security manager(“System.getSecurityManager() == null”) and if there is none then set a new one (“System.setSecurityManager(new SecurityManager())”). A policy file would also be required which could grant all permissions.

The system would be more realistic and user friendly if when clients the clients request to see all available auctions they would be shown how much time left each auction has before it closes. Another extension that would make the system more practical if the system would allow them to log in using a username and password rather than being provided an id.

Measuring the load and the throughput of the system would provide the system with a better performance evaluation as well as enhancing the fault tolerance of the system because the system would be aware when it's reaching its maximum capacity and in that case instead of throwing exceptions such as out of memory it could either inform the client that it cannot accept any more commands. It could then delete all closed items in order to allow clients to create new ones.